



Masterarbeit

Improving Data Locality in
Distributed Processing of
Multi-Channel Remote Sensing
Data with Potentially Large
Stencils

Philipp Patrick Posovszky



Masterarbeit

Improving Data Locality in Distributed Processing of Multi-Channel Remote Sensing Data with Potentially Large Stencils

Philipp Patrick Posovszky

Aufgabensteller: Prof. Dr. Dieter Kranzlmüller

Betreuer: Pascal Jungblut
Roger Kowalewski
Marc Jäger (DLR e.V.)

Abgabetermin: 06. February 2020

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 06. Februar 2020

.....
(*Unterschrift des Kandidaten*)

Abstract

Distributing a multi-channel remote sensing data processing with potentially large stencils is a difficult challenge. The goal of this master thesis was to evaluate and investigate the performance impacts of such a processing on a distributed system and if it is possible to improve the total execution time by exploiting data locality or memory alignments. The thesis also gives a brief overview of the actual state of the art in remote sensing distributed data processing and points out why distributed computing will become more important for it in the future. For the experimental part of this thesis an application to process huge arrays on a distributed system was implemented with *DASH*, a *C++* Template Library for Distributed Data Structures with Support for Hierarchical Locality for High Performance Computing and Data-Driven Science. On the basis of the first results an optimization model was developed which has the goal to reduce network traffic while initializing a distributed data structure and executing computations on it with potentially large stencils. Furthermore, a software to estimate the memory layouts with the least network communication cost for a given multi-channel remote sensing data processing workflow was implemented. The results of this optimization were executed and evaluated afterwards. The results show that it is possible to improve the initialization speed of a large image by considering the *brick locality* by 25%. The optimization model also generate valid decisions for the initialization of the *PGAS* memory layouts. However, for a real implementation the optimization model has to be modified to reflect implementation-dependent sources of overhead. This thesis presented some approaches towards solving challenges of the distributed computing world that can be used for real-world remote sensing imaging applications and contributed towards solving the challenges of the modern *Big Data* world for future scientific data exploitation.

Contents

1	Introduction	1
1.1	Task and Definition of Goals	3
1.2	Structure	3
2	Processing of Large Scale Multi-Channel Remote Sensing Data	5
2.1	Remote Sensing Images and Processing Images	5
2.1.1	Efficient Management of the Large Data Volumes	6
2.1.2	Loading and Distribution of <i>RS</i> Data	6
2.1.3	Irregular Data Access on Parallel File Systems	6
2.1.4	Complex Dependencies between Tasks and Data	6
2.1.5	Efficient and Productive Programming of <i>RS</i> Applications on Dis- tributed Systems	7
2.2	Hardware Limitations	7
2.3	Distributed Computing	9
3	Distributed RS Image Processing	13
3.1	Problem Statement	13
3.2	Data Description	14
3.3	Stencil Computation	14
3.4	DASH - a PGAS Framework	16
3.5	Patterns in DASH	16
3.6	Multi-channel RS Processing Workflow	18
3.7	Brick and Data Locality	20
4	Optimization Model for Data Locality	23
4.1	Parameter Definition	23
4.2	Surface-to-volume and height-to-width Ratio	24
4.3	Costfunction	25
4.4	Optimization Model	30
4.5	Optimization and Worker Software	30
4.5.1	Locality Optimizer for Remote Sensing Data	30
4.5.2	Remote Sensing Image Distributor and Processor	31
5	Results & Discussion	35
5.1	System Description	35
5.2	Performance Evaluation	37
5.2.1	Read/Write Speed	37
5.2.2	Speedup	37
5.2.3	Efficiency	37
5.2.4	Strong and Weak Scaling	38

Contents

5.3	Experiments	38
5.3.1	Brick Locality	38
5.3.2	Locality in Multi-Channel RS Image Processing with Potentially Large Stencils	42
5.3.3	Layout Performance with Different Stencils	46
5.3.4	Weak/Strong Scaling	57
5.4	Discussion	60
6	Outlook - Interaction with Python	67
7	Conclusion	69
	Symbols	73
	List of Figures	75
	List of Tables	79
	Bibliography	81

1 Introduction

Remote Sensing (RS) imagery provides an important source of information about our planet e.g. allowing to monitor traffic of ships in the ocean or detect oil spilling [Vel16], to identify damage on buildings after earth quakes [YIL⁺13], to detect deformation and seismic activity of volcanoes [JDT⁺15], and to generate *Digital Elevation Models* (DEM) of the Earth [ZMB⁺16]. These applications emphasize the huge significance of *RS* imagery in business, society and research today.

The information required for these applications is provided by spaceborne as well as airborne systems which generate a large amount of data on a daily basis. In total more than 200 orbital sensors on weather satellites, space telescopes, and observation orbiters capture remote sensing data of the Earth [MWW⁺15]. An example for such a system are the satellites from the *Copernicus Sentinel-2* mission from the *European Space Agency* (ESA) whose optical sensors cover the whole Earth every 5 days. The data collected between 2015 and December 2018 accumulates to a total of 6.7 petabytes (PB) which are structured in more than 13 million products [Sen18]. The sheer volume of the sensor data is a great challenge in terms of storage, management, processing and analysis a challenge. The *German Aerospace Center (DLR)* approached these challenges in its *TanDEM-X* mission by a fully automated process that achieved to generate a *DEM* of the Earth out of more than 500,000 data sets [ZMB⁺16].

This illustrates that the processing of *RS* imagery can be defined as a Big Data task. Because the term Big Data is relatively new and there is no exact definition, this work will refer to it using the 4 V's definition: Volume, Variety, Velocity, Value [HJ13]. The increasing *volume* of data caused by more and more satellite missions that provide a growing *variety* of *RS* imagery with increasing *velocity* can be used for greater *value* applications [HSHH15]. Table 1.1 lists exemplary volumes and velocities of current satellite missions.

Because of the growth in data volume and the increasing amount of observing spaceborne and airborne missions, it seems valid to define *RS* as a Big Data discipline. It covers all of the 4 V's of Big Data: (1) A large volume of data from the global missions. (2) A variety of data from different sensors. (3) An increasing value of the data for emerging businesses. (4) A high velocity of the data transfer from the sensors to the scientists. The latter is especially boosted by the change from a traditional order-request producing mode to an on-line data-triggered producing mode, which allows the scientist to have real time data access [MWW⁺14]. Handling the 4 V's is a challenging task that requires modern hardware systems, high-performance data management approaches and efficient processing algorithms.

The *Microwaves and Radar Institute (HR)* of the DLR in *Oberpfaffenhofen* with its *TerraSAR-X/TanDEM-X* mission and additional airborne platforms is one of the main players in generating and processing a large amount of *Synthetic Aperture Radar* (SAR) *RS* data. Figure 1.1 illustrates a data sets generated by a *SAR* sensor and the multi-temporal dimensionality. These data becomes increasingly extensive as new sensors and systems are developed. With more channels as well as a higher resolution of the imagery, the data volume will likely exceed more than 1 terabyte (TB) per data set in the future. Handling such

Satellites	Velocity (Mbps)	Volumes (GB/Day)	Volumes (TB/Year)	Year
HJ-1B	60.00	57	20.32	2015
HJ-1A	120.00	114	40.64	2015
[...]	[...]	[...]	[...]	[...]
LANDSAT5	85.00	28.02	9.99	2015
RADARSAT-2	105.00	57.68	20.56	2015
LANDSAT8	440.00	241.70	86.16	2015
TanDEM-X*	40.00	140.00	50.00	2018
SENTINEL-2	2933.00	4124.53	1470.17	2018
Total	30143	4620	1648	

Table 1.1: Satellite data center: the volume and velocity of RS data [MWW⁺15], [Sen18], (*Projected based on [RMW⁺18])

large data sets is a challenge in itself.

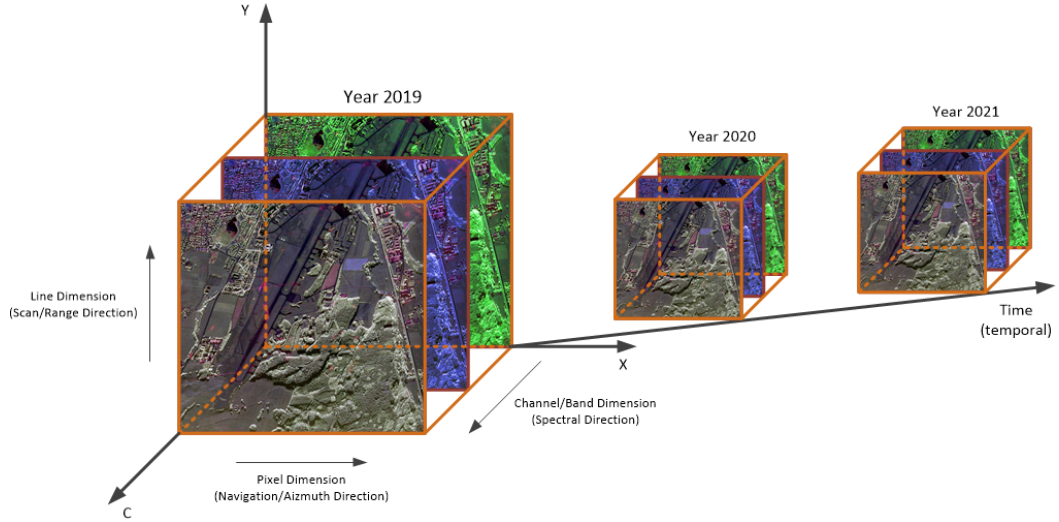


Figure 1.1: The dimensionality of the multi-temporal *RS* data.

Current techniques used to process and analyze *RS* data are time consuming and expensive in terms of computational costs and related infrastructure [Vel16]. Because of optimizations of the execution time, the *Input/Output (I/O)* times become critical factors for the economic and scientific value of the processed data. This is particularly important for applications where a time-criticality is a requirement, such as the monitoring of the soil moisture of farmland for finding places to irrigate or oil spills to hunt the polluting ships.

Parallel programming promises a significant speedup of the processing in data-intensive *RS* applications. Frameworks like *Message Passing Interface (MPI)* [CGH94] allow to distribute

the computational costs to distributed systems for a faster execution of the algorithms. Nevertheless, the required handling of multilevel hierarchies and increasingly complex distributed systems turned out to be difficult and error-prone [MWW⁺14]. A variety of frameworks on the market exist which proclaim to solve the issue to handle complex implementations with *MPI*, such as *UPC* [UPC05], *Titanium* [YSP⁺98], *Chapel* [CCZ07] and *DASH* [FFK16]. These frameworks provide an abstraction layer taking care of e.g. data synchronization to simplify the algorithm development. The possible speedup of existing *RS* algorithms using parallel programming combined with the simplified development using a *MPI* abstraction framework makes the investigation of a respective solution a worthwhile project.

1.1 Task and Definition of Goals

DLR's Microwaves and Radar Institute investigates new approaches for the processing of raw *SAR* data and high level products in a reliable and fast way. One of the goals is to speedup the processing time to be able to produce more and faster high level products on a big scale, like the *DEM* from the *TanDEM-X* mission [ZMB⁺16]. The respective *RS* imagery is processed requiring large stencil operations, see for a definition in Section 3.3 in Chapter 3. In the *RS* image processing such stencils are required to solve dependencies between several pixels in the image, e.g. to focus the image. These operations are similar to normal image processing operations like a smoothing filters, but with a way more larger extent in the stencil dimensions. Normally the stencils are in a rectangular shape. These large stencils make it difficult to parallelize the processing of the algorithm because of the data synchronization between the different nodes in the distributed system.

This work evaluates the improvement of the data locality and the data layout in a distributed environment for the processing with potentially large stencils in order to reduce the communication overhead. Therefore the impact of large stencils on the communication and network saturation is analyzed and possible hardware bottlenecks in terms of network capacity and disk *I/O* are identified. Furthermore, a possible distribution of the data on disk aligned to the *DASH* patterns is investigated. At the end of the work the new approach towards optimize network traffic of the processing queue of a multi-channel *RS* data processing with prior knowledge of the stencils is evaluated. The goal is to minimize the network traffic with exploiting the locality on disk and the data layout in memory with respect to the given processing task.

1.2 Structure

The remainder of this work is structured as follows: Chapter 2 introduces the theoretical background of this work. It covers an introduction to the basics of *SAR* data and the processing of multi-channel data. Furthermore, a small excerpt of the current state of the art in distributed computing of *RS* data is summarized.

Based on this, chapter 3 defines the problem investigated in this work as well as the structure of the data used throughout this work. In addition the *Portal Global Address Space (PGAS) High Performance Computing (HPC)* programming models are described together with an introduction to the *DASH* framework for parallel computing.

Chapter 4 covers the optimization approach of this work. An overview over optimization strategy and the cost calculations are given. The chapter also describes the foundations of

the approach and the resulting implemented software. The software is split in one part for creating an execution plan, written in *Python*, and another part for processing the images, written in *C++*. The results of the evaluation experiments are presented and discussed in chapter 5. Afterwards, a short envision of a potential *SAR HPC* Processing Python API based on *DASH* is shown in chapter 6. The work is concluded in chapter 7.

2 Processing of Large Scale Multi-Channel Remote Sensing Data

Multi channel data sets have become important in many fields of science. The mankind is developing better methods to measure physic abilities with increasing rapidity and so also the data sets are growing in dimensionality and size, see Figure 1.1. Especially the earth observation scientists were able to increasing the data amount in the last decades with more and cheaper satellites. Handling this amount of data is a huge challenge. In the next section an overview about the challenges in working with a large amount of *RS* data is given. In the follow section the physical limits of processing on new hardware developments is shown. The chapter concludes with the introduction of the distributed computing concept for *RS* image processing.

2.1 Remote Sensing Images and Processing Images

In the last years various new approaches to handle *RS* data on distributed systems where initiated resulting in growing research in the field of *RS* Big Data. Many scientists are working on solutions for the *RS* Big Data issues that allow future researchers to successfully analyze the huge amount of available *RS* data. In addition to that, large companies generate new businesses out of the emerging data and invest in research for mastering the mountain of data. With the open access policy of *ESA*'s Copernicus mission data, Google join the stage with its cloud platform *Google Earth Engine* which provides a platform with simple abstract access to the data and globally distributed computer centers in the background [GHD⁺17]. One of the first results on a global scale is an urban footprint layer generated from hundred of thousands of Landsat and Sentinel-1 scenes on the Google Earth Engine cloud platform [GMÜ⁺17]. In the future these data and the developed processing techniques will be useful in various kinds of applications with great value for humankind. For example it will be possible to monitor the growth of a field with satellite images [LSBBH11], measure the soil moisture to enhance the yield of harvest [BAZ12], or monitor the environmental pollution of the ocean by ships in real time [Vel16].

The overview paper “Remote sensing big data computing: Challenges and opportunities”[MWW⁺15] points out that the field *RS* Big Data leads to five main research issues:

- Efficient management of the large data volumes.
- Loading and distribution of *RS* data.
- Irregular data access on parallel file systems.
- Complex dependencies between task and data.
- Efficient and productive programming of *RS* applications on distributed systems.

In the remainder of this section the five research issues are described in more detail and some examples for solutions are given.

2.1.1 Efficient Management of the Large Data Volumes

Managing the rapidly growing amount of *RS* data is a difficult challenge especially as new globally distributed data centers arise resulting in the first and second issue. First, the data centers should have easily accessible systems, so that scientists can query the metadata and order/process data in a fast way. This in combination with the multi dimensionality of *RS* data, see Figure 1.1, makes storing and accessing distributed *RS* data complex. Second, the synchronization between data centers or the transfer of the data to the scientist's processing resources takes a lot of time. In the best case the data would be processed directly on the storing location which results in a data center which have to provide storage and computing power together. Provide storage, computing power and the common infrastructure together with the concepts to store the data efficient and manage them is expensively and can only be achieved by large players.

2.1.2 Loading and Distribution of RS Data

The computation of each pixel usually depends on a large amount of pixels in the neighborhood or data from another spectral band, see Section 3.6. Through this dependencies it is not easily possible to decompose the data in several data chunks and computed independently. If the data is distributed to many nodes and the low-level *MPI* is used, this could result in repeated calling of *MPI* send/receive communication which results in a significant performance decline. This is a main concern in this thesis and is deeper described in Section 3.7.

2.1.3 Irregular Data Access on Parallel File Systems

The complex dependencies between tasks and data. An example therefore is scientist uses data from a radar satellite and an optical satellite from different data centers, or just processes a time series, so the data still have to transfer and be accessible in a fast way on the local *file systems (FS)* [MWW⁺15]. This is addressed in [WMZ⁺15], accessing multiple files could be a performance issue for a parallel file systems, like *General purpose file system (GPFS)* from *IBM* [IBM19], which are used in modern environments and are often optimized for contiguous file access. [WMZ⁺15] shows one promising approach towards solving the *I/O* issue is the use of a modified *OrangeFS* file system with application-aware data layout policies for *RS* image processing. A 20 to 30 percent *I/O* performance improvement was obtained, mainly by segmenting the *RS* image into multiple bricks and use a Hilbert Curve to distribute them on the *I/O* servers. The data inside the brick was ordered with Z-order curves.

2.1.4 Complex Dependencies between Tasks and Data

To solve specific research tasks the fusion of many different data set is necessary, e.g. in *RS* to generate a image it have to be geo-referenced to the correct position of the earths surface which requires several other data sets, like the airplane flight trajectory/satellite orbit data. Also the results from a specific task could be depend on other task, which could be make

the whole processing chain quite complicated. [SBL⁺17] shows examples for solving the dependencies between different data and reduce the complexity with a *data-cube*.

2.1.5 Efficient and Productive Programming of RS Applications on Distributed Systems

The efficient and productive programming of *RS* applications on distributed systems depends on all four previous mentioned research issues. (1) Without an easy way to integrate data access in the to developed applications it is not possible to process data in a fast way. (2) Decompose the data in several chunks, process these data on a distributed system with thousand of nodes and summarize the results is much more complicated than just process everything on one single node. (3) the common file systems are not aligned to the irregular data access. (4) data fusion of distributed data on top results in a complicated data handling and fusion. Another problem is the different system architectures that exist today: from personal computers and servers to huge clustered servers and supercomputers, each with its own challenges. [PDCK11] investigates the trend in *RS* distributed computing to use retired personal computers with an *MPI*-based application to distribute the work showing that in such heterogeneous platforms additional considerations towards load balancing are necessary. The article also compares applications for such a personal computer cluster with applications written for *Graphical Processing Unit (GPU)* or *Field Programmable Gate Array (FPGA)* systems [PDCK11]. This different architectures of processors must also be considered in terms of their ability to efficiently process distributed remote sensing data. In the paper “High performance GPU computing based approaches for oil spill detection from multi-temporal remote sensing data” [BDK⁺17] a significant speedup was demonstrated by using *MPI* with 64 cores for attribute computations. But still this was far away from proving real scalability of these effects on huge *High Performance Computing (HPC)* systems. Generally speaking, a good scalability means e.g. that more speed is achieved when using more resources, for a more detailed definition see [Bon00]. At best, the amount of resources added correlates to the increase in speedup, but this is not achievable in the real world due to *Amdahl’s Law*, see Section 5.2.2.

All this together makes it difficult to implement efficient programming for distributed processing of distributed *RS* data. But this does not only apply to *RS* images distributed processing, most of the points are also applicable to *Big data* in general.

2.2 Hardware Limitations

Powerful computer systems are needed to process the large amount of *RS* data. Today there are different kinds of computers, e.g. with up to 24 cores per CPU and up to 1 TB RAM [Int19], file systems up to exabyte and fast network interconnections. They are usually all structured according to the same scheme: “The von Neumann Computer Model” shown in Figure 2.1. This scheme consists of the *Central Processing Units (CPU)*, the memory, and the devices for *I/O*. These components are connected using the *System bus* consisting of a *Address bus*, *Data bus* and *Control bus* [von93].

CPU’s became more powerful in each technology iteration of the last decades. In the last 40 years the performance of microchips increased by the factor of 10,000 since 1978, see Figure 2.2. This development led to the postulation of the two laws: (1) The *Dennard Scaling* postulates the possibility to keep the current/voltage dropping and maintain the

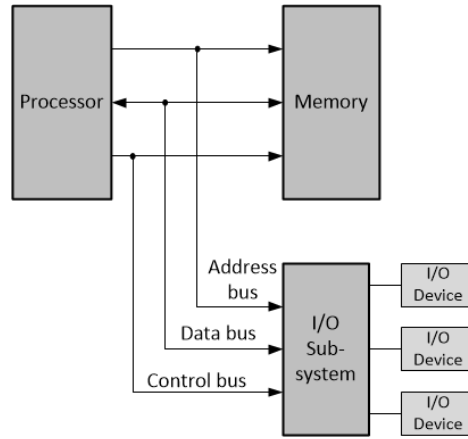


Figure 2.1: Schema of von Neumann Computer Model

dependability between integrated circuits. (2) *Moore's Law* predicts the doubling of the number of transistors per chip every year. But the performance increase per generation is falling with each new development. The flattening of the performance increase curve in the last 15 years visualizes the end of the two laws. Around 2004 the *Dennard Scaling* ended and more recently in 2011 the *Moore's Law* has been shown to be no longer valid [HP11]. At this point more and more CPUs with parallel cores come to the market and start to bring a further performance boost, but only if the application is developed for parallel execution.

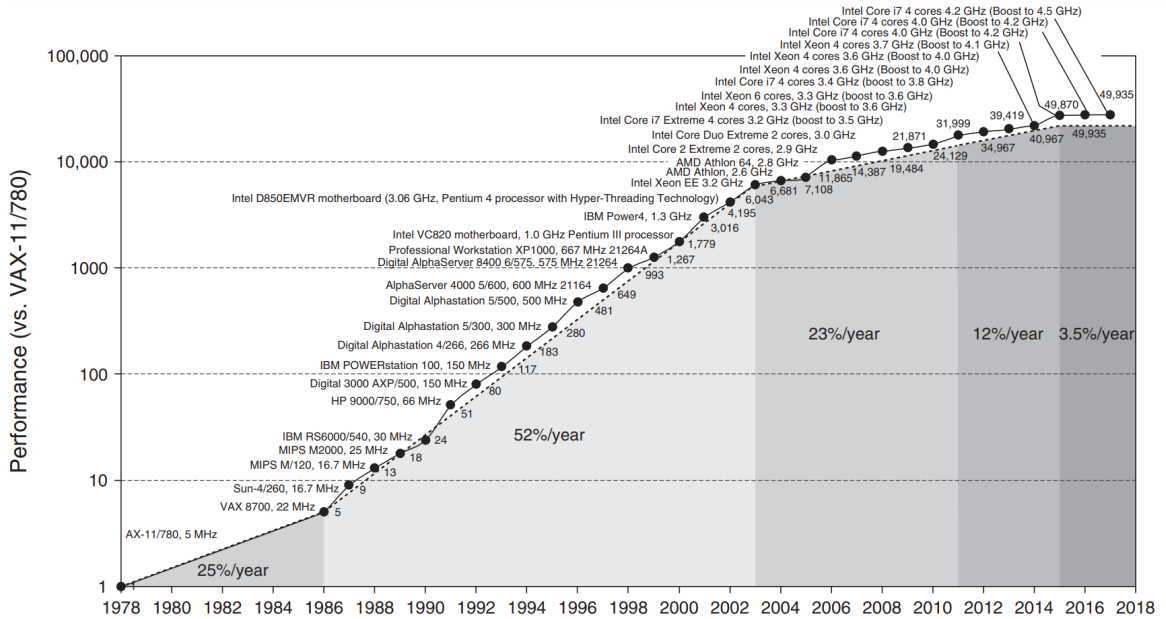


Figure 2.2: Growth in processor performance over 40 years [HP11]

The main concern is the *I/O* subsystem for processing *RS* data, see Seciton 2.1.2 about the research issues. Modern cluster file systems like *GPFS* [IBM19] allow to manage theoretically more than 633,825 yottabytes of data. So the main aspect limiting the size of the FS are the

financial resources available to invest in the infrastructure. This could be quite expensive even though the cost-to-disk-space ratio has been continuously improved in the last years. But not only the size matters, but also the way to manage and access the data. As already mentioned in the previous section this could become a problem with processing *RS* data sets which use irregular data access patterns and are large in size itself. Even if continuous reading/writing provides good results, irregular data access can be slow and thus a potential showstopper for scaling up. Also the network interconnection is a limiting factor. First of all the delay/latency while accessing data over a network is much slower than accessing a local solid-state disk directly, because of the communication through the hardware stack from the *Processor* to the *I/O Subsystem* over the *I/O Device* to another system and back again takes long then access the local *I/O Device*. The communication over the network with the other system could exceed the available resources, which quickly ends in a bottleneck. Adapting the network structure to cope with these problems has proven to be very costly.

As previously described it is no longer possible to rely only on hardware enhancements in the microchip architecture of general purpose systems to speed up throughput in general. One solution to gain more speedup is the development of a *domain-specific architecture* that is optimized for a specific *RS* task. This approach promises a system that is highly efficient in processing specific *RS* task but does not scale to arbitrary tasks. This makes such an specific architecture expensive and maintenance intensive [HP11]. Another approach towards an optimized processing speed is the focus on code optimization in general. This is particularly cost-intensive in terms of time, personal resources and still does not solve the problem of hardware limitations.

The DLR's Microwaves and Radar Institute is currently using powerful servers for data intensive processing. These servers are equipped with a large amount of memory allowing to process extensive *RS* data sets. But to deal with the amount of data generated from the new technology *digital beam forming SAR (DBFSAR)* [ABA⁺17] system, even bigger data sets need to be handled which will start to exceed the local memory of a single node. Distributing the processing algorithms to a variety of nodes is not trivial because most of the algorithms in the institute were developed for single node processing. The scientists currently start working on parallel solutions but there is still limited experience on this topic yet. One exception is a newly developed software that processes data via the *SPARK* [Apa20] framework, but an infrastructure to unleash the true potential of the *SPARK* framework was still not put into production.

Another solution is to use other new processing paradigms like distributed/high performance computing. In the following thesis the focus is to overcome hardware limitations with distributing the problem solution. The focus is on the improvement of data locality per processing node in the data initialization phase and improving the memory locality of the *PGAS* memory space.

2.3 Distributed Computing

Distributed computing combines the resources of multiple nodes to solve a computational task. Therefore the task is divided in smaller chunks which are processed on multiple participating nodes at the same time. The workflow is managed by a server node which distributes the work chunks to client nodes which process the work packages and return the result back to the server [BCPS13]. In Figure 2.3 a simplified schema of the described work process is

shown. A master node receives a list of tasks and distributes the tasks to the available nodes for computing. After all tasks were finished the intermediate results are all communicated back to the master and then merged to a final result. The master may use a database to store the progress or intermediate results.

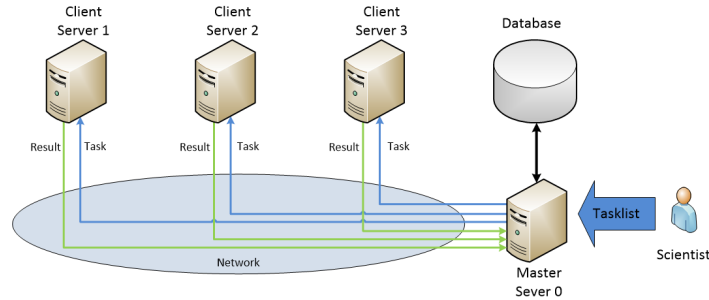


Figure 2.3: Simple abstract schema of a task distribution on distributed system with four nodes.

In the last decades a variety of cluster-based *HPC* systems appeared. One of the first projects heading towards the exploitation of clusters in *RS* was the *NEX* system of NASA, with 16 identical personal computers, with 100 MHz CPU clock speed, which are connected over Ethernet hubs to form a network [LGP⁺11]. This so called *Beowulf* cluster was able to process its task in half of the time compared to single node processing. Today exist systems with hundred thousands of cores, like the *SuperMUC-NG* at the *Leibniz-Rechenzentrum* in *Munich* which consists of 311,040 compute cores with 719 TB memory, and a peak performance of 26.9 PetaFlops/s in the *High Performance LINPACK* benchmark [Rec19]. With this amount of computational power it reached the 9th place in the world ranking of *HPC* systems, at the time of June 2019 [Pro19].

These cluster-based *HPC* systems are often used for the processing of complex scientific data: Scientists use them to simulate the behavior of their models. Astrophysics use it to simulate the fusion of neutron stars or super novas. Chemistry and material scientists use *HPC* for research in quantum matter or the simulation of oxidation processes on surfaces. Earth, climate and environmental scientists simulate the world climate or develop new weather forecast algorithms. Most of these simulations are computationally expensive, but some of them are also starting to become *I/O* intensive as well. One example is from the paper [BKBB18] the 4-D urban mapping project computed on the *HPC* system of the *Leibniz-Rechenzentrum* in *Munich* which already uses *RS SAR* products from the *TerraSAR-X* satellite mission of DLR.

The general benefits of a distributed computing approach compared to a parallel application on a single node are: more resources and more possibilities to achieve scalability, better reliability, data sharing and combining heterogeneous systems. Thus benefits are at the expense of challenging system development, as mentioned in the previous section. It is technically easy to add more nodes to the system in order to increase the processing speed. But in practice, the parallel efficiency drops with the increase of CPUs, obeying the *Amdahl's law*, see Section 5.2.2. One reason for this is the communication demand between the nodes while solving the computation, called the communication overhead of the algorithm. The amount of communication overhead is strongly related to the algorithm itself, problem size, number of participating cores/nodes, and data distribution. With every core that is added

to the system, the amount of work per node and the part of problem it holds locally on memory is decreasing and can be solved faster. But at the same time, especially in the case of multi-channel *RS* image processing with a potentially large stencil, the communication overhead drastically increases, because of more dependencies to neighbor cells on a remote node. This problem is detailed in section 3.

As the *RS* data sets get bigger and bigger, the *I/O* operations will start to become a big part in the total execution time in distributed systems. To this day, most of the petascale supercomputers worldwide are not good at loading or transferring this huge amount of data. A reason for this is that locality optimization in the data storage architecture has been no main concern for such supercomputers in the past [MWW⁺15]. As a novel approach the *Leibniz-Rechenzentrum* in *Munich* start in 2020 a *Data Science Storage (DSS)* which should solve the demands and requirements of data intensive science [Rec19].

Distributed computing is important for the processing of the growing *RS* data sets. A major limitation today is that current *HPC* systems are not designed for the high *I/O* load that this type of data causes. This thesis studies an approach to account for these limitations by using the *I/O* of a local node and optimize the data distribution to fit to the *PGAS* memory layout. This should reduces the communication overhead so that the system scales better when used with multiple nodes. As a result, the use of a *Beowulf* cluster consisting of old personal computers or of different servers with different CPUs could become feasible for *RS* data processing in the *HR* Institute.

3 Distributed RS Image Processing

Achieving linear scale up in speed and processing is a difficult challenge. It depends on various factors, e.g. on the used algorithms, problem size, amount of processors, data layout, cache sizes and so on. The following section gives a more detailed description of our problem. It is followed by the technical background and description of the used framework.

3.1 Problem Statement

The goal is to investigate what are the effects of decompose a *RS* image and process them afterwards with potentially large stencils. Effects in terms of network, image reading speed from local hard drives and overall processing speed. The reason for using distributed computing is for a large data set with more than 1 TB, systems using a *Non-Uniform Memory Access* (NUMA) architecture [BFS89] are at their limits in terms of computational performance and *Random Address Memory* (RAM) utilization. Most of the common nodes today, such as the *Xeon Gold 6142* [Int19], are limited by a *RAM* capacity of 1 TB. Planned future releases will allow up to 4 TB. But even if the problem fits into the *RAM* of a single node, only a rather small number of cores can work on the data thus limiting the processing efficiency. Distributed or parallel computing could solve this issue by allowing many individual nodes to participate in the processing by sharing their resources. But with this kind of processing other problems arise. We have three key issues while transferring or writing a program for distributed or parallel computing: (1) Existing programs using a variety of computation types, structures to solve problems and this is not easy to transfer to an uniform approach. (2) The variability of the computational resources has to be addressed because the nodes could have a dissimilar amount of load or different computational capabilities. (3) The communication overhead has to be taken into account because inter-node communication over network is slower than inter-machine communication via the internal bus [Bok87].

For this thesis the focus is set on the third issue while using distributed computing. The first issue is mainly a problem for migrating existing applications to distributed computing. The dissimilarity in load and computational capabilities will only be a problem if there are competitive calculations in parallel or if one node is much slower. The communication overhead occurs when the processing on the nodes is not independent and intermediate results have to be exchanged to other participating nodes. This could be of particular interest for *RS* data operations with a huge dependency to neighborhoods using potentially large stencils on large arrays, as described in detail in the next section.

One way to minimize the communication over the network in the initializing step could be to increase the locality of data in the local storage such its aligned to the *PGAS* memory layout such that the demand of following processing steps with different stencils is satisfied at its best. This means in the best case, each node holds the data which are supposed to be in its part of the *PGAS* memory layout while the processing directly on it's local disk, rather than loading it over network.

3.2 Data Description

The *SAR* data sets are structured similar to an ordinary optical image. The whole image consists of a two dimensional pixel array. Each pixel is defined by its position x, y and a value n for a specific physical property, e.g. the color in the case of an optical image. However, in a *SAR* image n specifies a complex number that represents the amplitude and phase of the electromagnetic wave. In an optical image n often consists out of multiple values that specify the position of the appropriate color in a color space, e.g. the *RGB* color space is defined by three channels for the light intensity of red, green and blue. A *DBFSAR* image also can have up to 60 channels per image and more [ABA⁺17]. If we assume a image with $100,000 \cdot 25,000$ pixels and one pixel being represented by a 8 Byte complex number, which is common for a *SAR* image, with 60 different channels, the resulting size is 18.62 GB. Each of these channels could be processed separately and combined for different kinds of high level products. *RS* images normally contains a stack of metadata, e.g for a airborne system the *flight trajectory*. For this thesis the metadata are not considered.

For loading the data on different nodes in parallel they have to be segmented into bricks, like in the Figures 3.1 and 3.5 shown. There the whole image is segmented in regular quadratic blocks, called a data brick. Its also possible to segment the bricks line or column wise, to match with the later described *DASH* patterns. The position of the brick could be encoded with the x, y coordinate of the upper left and lower right coordinate. Each data brick will be distributed to a local *FS* of a participating node. A boost in *I/O* while reading/writing on the local disk of a single node in comparison to a parallel file system is expected. The reason is, at a certain point either the parallel file system, in particular the server, will not be able to handle more *I/O* request or the speed of the network interconnect becomes a bottleneck.

In chapter 5 evaluation experiments will be performed. Instead of generating the workload with real *SAR* data and algorithms, artificially generated grayscale images for the input and for the processing step a smoothing algorithm for simulating the workload of potential large stencil algorithms is used, see section 4.5.2. Real *DBFSAR* algorithms are not used in this thesis, because of they are not available in a compatible programming language for distributed computing, like *C++*, right now. Another requirement is the image width and height have to be divisible by powers of two.

Despite the approach with distributed solutions it would also possible for a large image to segment it in a lot of independent smaller images and stitch them back together after the processing. However, this generates a lot of more computational overhead and complexity for the whole processing.

3.3 Stencil Computation

Following [SSPP11], stencil operations are defined as: The neighborhood is commonly defined by a rectangle. A simple example for such a stencil operation is to add up all cells in the neighborhood and divide them by the total count to get the mean value of the neighborhood. A similar calculation is used by smoothing filters in image processing. In *RS* image processing, and the radar signal processing in particular, the neighborhood of an update cell often becomes extremely large which leads to a problem with communication overhead over multiple nodes, as stated earlier. Also, keeping all necessary cells in the cache of the CPU

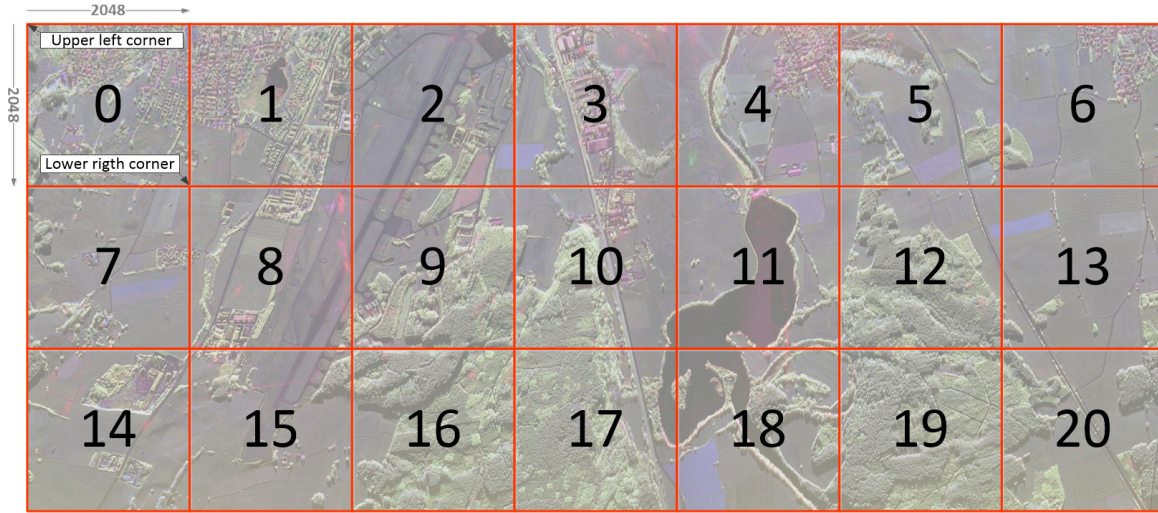


Figure 3.1: A *SAR* image segmented in 20 data bricks with a size of $2048 \cdot 2048$ pixels each.

for a single cell update will become difficult. Furthermore, as these kind of operations are of low arithmetic intensity, they hardly benefit from repeated access pattern on already loaded cache lines. This could, depending on the domain and stencil extents, generate a lot of cache misses which limits the total throughput to the node [SSPP11]. The performance of the real execution time of a stencils can become relevant for compare the different possible *PGAS* layouts and estimate costs for them.

Figure 3.2 shows an exemplary stencil operation for a $5 \cdot 5$ array. On the center pixel a stencil is applied, which smooths the array by calculating the mean of the $3 \cdot 3$ neighborhood as described earlier. The resulting mean value for the center pixel in the example is 15. The requirement in this thesis for the stencil extents is to be odd, otherwise it's not possible to have a center pixel. This is not true for radar signal processing, there are also even-sized stencils common.

1	23	243	33	33
54	1	9	3	3
6	23	3	33	33
1	24	34	1	1
45	24	4	3	77

Figure 3.2: $3 \cdot 3$ stencil operation on a small array.

3.4 DASH - a PGAS Framework

The following introduction to the *DASH* framework is mainly based on “*DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms*” [FFK16].

A *partitioned global address space* approach allows each participating nodes to have a global view on the data structure, e.g. an 2D array, inside and globally shared memory. Each node participate with a part of it's local memory to the globally shared memory. This (*PGAS*) memory space is distributed to all nodes and can be adapted by certain patterns in the layout. Thus, the application can benefit from a large shared memory, but additionally with the possibility to control the memory layout (global or on local level) to the demands of high performance and scalability [YBC⁺07]. The possibility to control the memory layout is a key factor for this thesis. In the following, a rough overview over *PGAS* approaches on the market and more details about the *DASH* concept are given. Next to *DASH* there are several other *PGAS* approaches on the market, like *UPC* [UPC05], *Titanium* [YSP⁺98], and *Chapel* [CCZ07]. But they all come as a new language, and thus with a compiler, which means a complete new ecosystem. Usually, this requires to train developers for the new language, which leads to high costs and only a few organizations can afford this. Also, it is unsure if the compiler developers continue their work and can give support in the future [FFK16]. With release of *C++11* [ISO12] and its introduced powerful abstraction mechanism for a generic, expressive and highly optimized library, many projects started to use *C++11*, like *UPC++* [ZKD⁺14], *Kokkos* [ETS14], *RAJA* [HK14], and also *DASH* [FFK16]. Like mentioned in Section 2.2 and Section 2.3, remote accessing and sharing memory becomes more important in distributed systems. For current and future large-scale systems *PGAS* is widely considered as a promising approach. The *DASH* framework uses a one-sided communication model, which is based on *MPI-3 Remote Memory Access (RMA)* features and is the basis for the runtime system of *DASH*, called the *DASH RunTime (DART)*. *DASH* itself is implemented as a *C++* template library. In this way there is no requirement for a custom (pre-)compiler infrastructure, this is called a compiler-free *PGAS* [SKF18].

This thesis follows [FF16] in the use of terminology when referring to *DASH* features:

- *unit*: single CPU core which contributes storage and processing resources.
- *teams*: several units can be organized in hierarchical teams.
- *patterns*: partition the global memory of an array into *blocks* and map them to a specific unit.

We have decided to use the *DASH* framework for testing the *HPC* ability of our experiment and to prove our hypothesis about improving locality on the disk while initializing a *PGAS* data array and optimized the memory layout for processing of multi-channel remote sensing data with potentially large stencils.

3.5 Patterns in DASH

The features of the *DASH* library can be used in order to improve the data locality and avoid communication overhead. The *DASH* library allows to use different variants of data distribution patterns for the shared *PGAS* memory which can be configured by specific parameters, e.g. parameters are the array size in the dimensions, the amount of used units,

see Listing 3.1. These patterns define how the global *PGAS* view is built up of the memory of all *units*. For each dimension it is possible to choose from *CYCLIC*, *BLOCK-CYCLIC*, *TILED* and *BLOCKED* pattern. If there is more than one dimension and a pattern is already chosen, it is also possible to specify *NONE* for the remaining dimensions. Listing 3.1 shows the instantiation of a *NONE BLOCKED* pattern. The amount of *units* is chosen by the *mpirun* call and inside the implementations the maximum amount of *units* is used. It is possible to group up the *units* in different teams and create a distributed array only for this team, but this feature is not used in this thesis. Per default the *units* are assigned to the given nodes using a round robin manner, though direct *unit* mapping is also possible. The version of *DASH* used in this thesis does not support overlapping memory areas at the edges, so that edge areas are automatically synchronized. In the next release of *DASH* this feature will be usable.

Listing 3.1: Instantiation of *NONE BLOCKED DASH* pattern

```
dash::TeamSpec<2> ts(1,dash::size());
dash::DistributionSpec<2> ds(dash::NONE, dash::BLOCKED);
dash::SizeSpec<2> ss(extent_x, extent_y);
dash::Pattern<2> pattern(ss, ds, ts);
```

So it is possible for the application developer to specify a data distribution which achieves a high data locality in the initialization phase and communication avoidance in the processing phase, depending on the chosen pattern and stencils. Examples for the different layouts are shown in the Figure 3.3. Each color shade represents an *unit*, the *units* can be placed on independent nodes. In the following thesis we use the abbreviation *B* for *BLOCKED* and *N* for *NONE*. As a synonym for pattern also layout can be used. Distribution could be used in both context, the *brick* on the disk or the *blocks* in the memory.

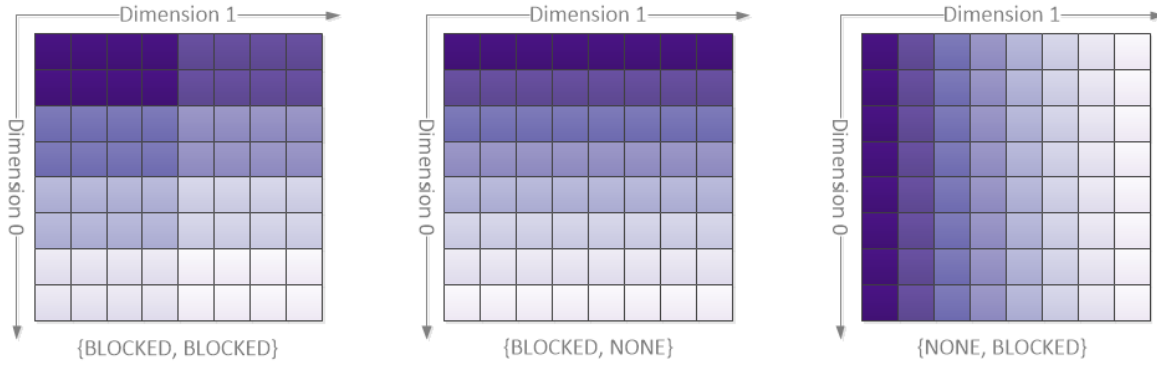


Figure 3.3: Different *DASH* patterns with 8 *units* on a segmented image in 8·8 bricks. Each color shade represents a block on an different *unit*.

3.6 Multi-channel RS Processing Workflow

The processing of *SAR* or multi-channel *RS* images is comparable to classical image processing, just with more channels and bigger stencils. In Figure 3.4 an exemplary workflow for multi-channel *RS* processing with eight different channels is illustrated. Transformations that only apply to a single channel are depicted on the left side of the figure. A high level product processing step combining different channels is shown on the right side of the figure. At the end of this exemplary workflow, six high level products are generated. An example for such a product is depicted in the background of Figure 3.1.

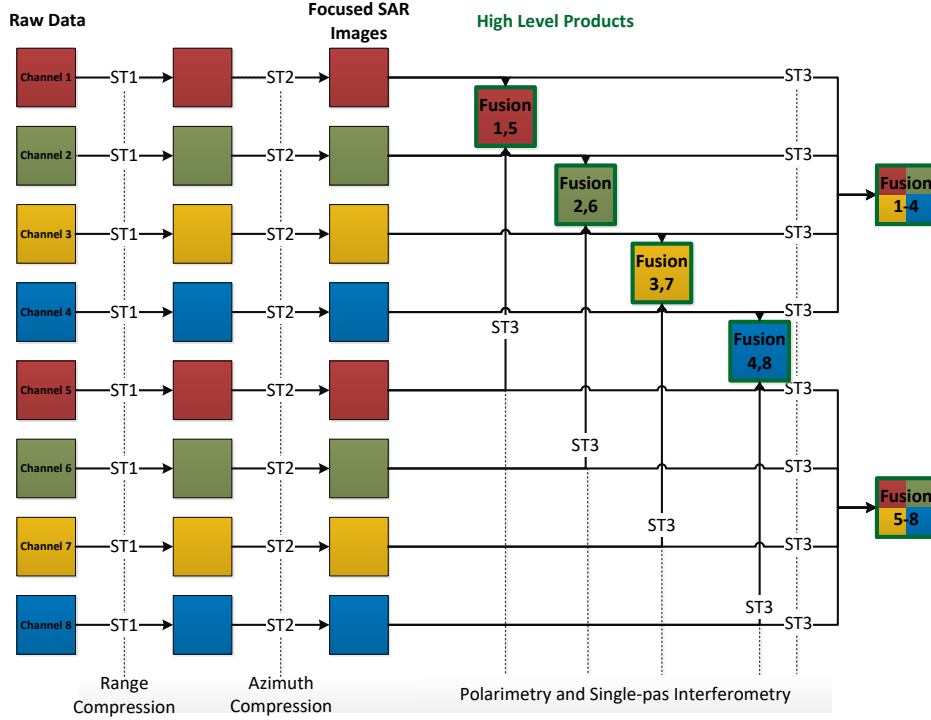


Figure 3.4: Multi-channel *RS* processing workflow

The arrows in the Figure 3.4 represent computations on the image. These computations are distinguished in computations on a single channel, called *Transformation* (T), and combinations of various channels, called *Fusion* (F). The transformation step preprocesses the channel in order to e.g. obtain equally illuminated scenes or focus the image in each dimension. Latter is needed to account for antenna movements along the flight track of airborne *SAR* systems. Fusion tasks combine different channels to generate high level products, like interferometric or polarimetric products. After each computation step the nodes have to write the intermediate results back to disk to keep them for follow-up processing. Transformations often use stencils with a big size in one of the dimensions (see *ST1/ST2* in Figure 3.5), e.g. to compensate the proper motion the airplane in the process of focusing the *RS* image in each dimension, in the case of a airborne *SAR* system. Compared to this, the

stencils of Fusion operations are more square like, as depicted by *ST3* in Figure 3.5. General the size of the fusion stencils is by a factor of 10 smaller compared to the other stencils which could have extents up to more than 1000 pixels in width or height. In contrast to the small stencils in classical image processing with e.g. sharpener filters with stencil sizes of $3 \cdot 3$ or $5 \cdot 5$, the stencils in *RS* image processing could become significantly larger.

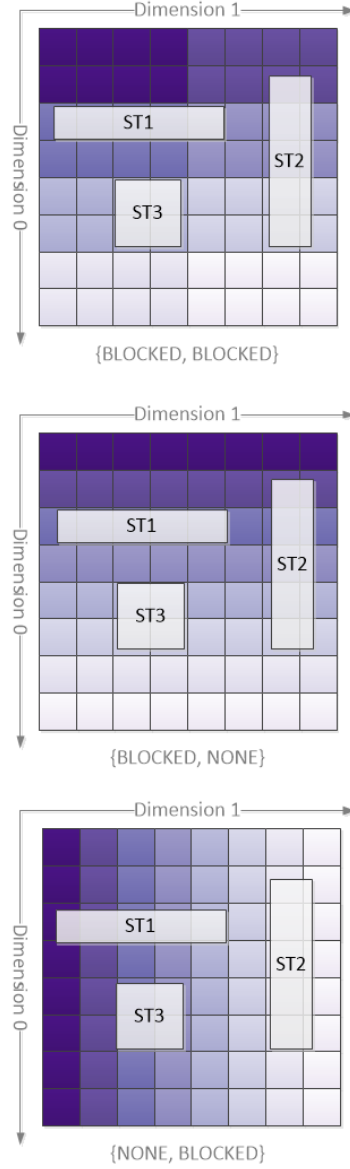


Figure 3.5: Different stencil operations: (ST1) with high extent in dimension 1, (ST2) stencil with high extent in dimension 0 and (ST3) stencil with small and similar extent in both dimensions. Color shade represent a *unit*.

3.7 Brick and Data Locality

The localities are distinguished between initializing the data from the hard disk into *PGAS* memory, called *brick* in this context, and during processing in RAM, called *block* in this context. A *brick* represents the data on the local disk as an $n \cdot m$ pixel array. A *block* represents an area in *RAM* of a *unit*. From this we distinguished between two locality concepts, *brick locality* and *block locality*.

Brick locality is defined as the amount of *bricks* which are stored on the node which will holds the *block* in its local memory after the initialization.

Block locality is defined as the amount of *pixel* which have not to be moved to another node while switching between two layouts.

In both locality definitions there is no network transfer necessary for initializing the data or switching the memory layout. In the Figure 3.6 an simplified example for *brick locality* is shown. A image is decomposed in $8 \cdot 8$ bricks and distributed to eight nodes and each node stores two rows of bricks. Also it is assumed each node handle the execution of one *units*, so each node is responsible for loading eight bricks. If the bricks are now initialized into an *DASH* array with the *BB* pattern, a 50% *brick locality* is achieved. Using a *BN* pattern matches the storage pattern, which is row wise on the nodes, thus increasing the brick locality to 100%. The other way around with a *NB* layout, results in only 12.5% *brick locality*. The bricks which are local stored on the node disk are highlighted red in the Figure 3.6. With a higher *brick locality* factor less bricks have to be copied to another *unit*, which result in less network load. If the memory *DASH* pattern exactly matches the storage brick pattern, then there will be no remote calls at all and the system can read all memory from the fast local disks. With no remote calls the total *I/O* speed will be no longer depending on the total throughput of the network and storage infrastructure. Instead the accumulated disk speed of all nodes is the maximum *I/O* which can be achieved. If *brick locality* is decreasing, the network will be again also an factor to consider.

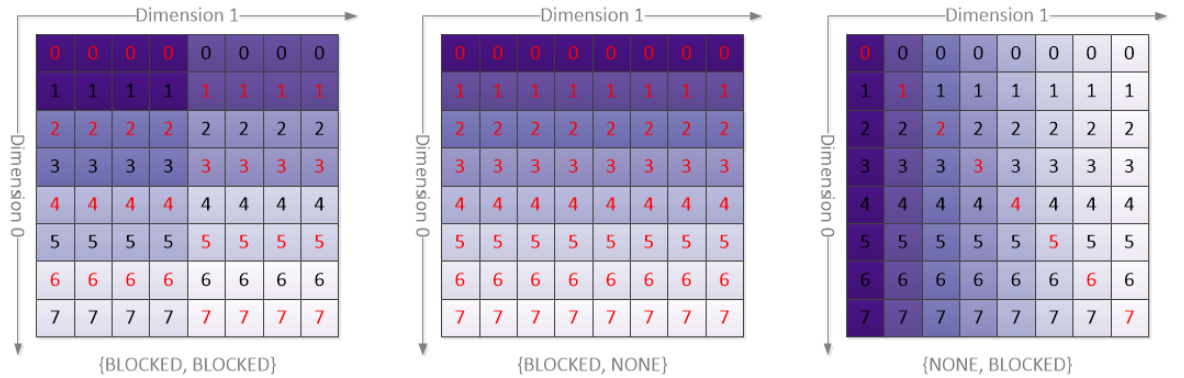


Figure 3.6: Visualization of an image segmented in $8 \cdot 8$ bricks loaded into a *BB*, *BN* and *NB* patterns in *DASH* using eight *units*. The different block colors shades correspond to the different *units* and the brick number corresponds to the respective node. Red highlighted brick numbers are locally available on the disk.

Multi-state processing pipelines may benefit from a switch in the *DASH* layout between individual processing stages. The amount of traffic generated by switch the layout and processing on a aligned stencils could be less then the traffic is processing with an unaligned

stencil. In the Figure 3.7 an simplified example for *block locality* is shown. Given a *DASH* array with a *BB* pattern consist out of $8 \cdot 8$ bricks. If a layout switch is assumed to a *BN* pattern 50% block locality is achieved. The other way around, for the *NB* pattern 12.5% is achieved. In contrast to in the *brick locality* example, some *units* have no data in it's local memory at all. While switch a layout the network decreases with a higher *block locality*, because less data have to be communicated to other *units*.

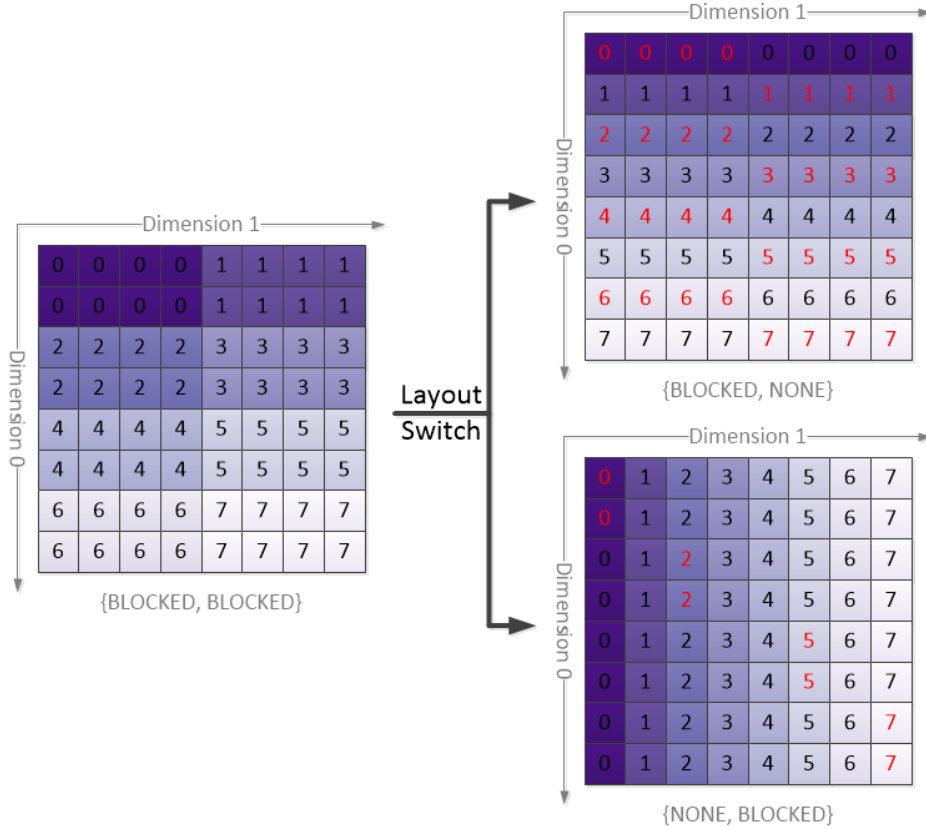


Figure 3.7: Visualization of an image segmented in $8 \cdot 8$ and already loaded into a *DASH* array with *BB* pattern. The different block colors shades and numbers correspondent to the different *units*/nodes. Red highlighted brick numbers are available in the local memory.

4 Optimization Model for Data Locality

The aim is to minimize the total volume of inter-node communication. For this purpose it is necessary to consider the amount of communication that is required to initialize the bricks into the memory as well as the amount of communication due to stencils processing. This can be achieved by aligning the brick distribution on the nodes to the *DASH* memory layout in the initialization step and additionally align the memory layout to the used stencils.

A processing chain is divided into multiple steps as described in Section 3.6 (Fig.3.4). At the beginning the data is read into the local memory from single bricks which are distributed over all nodes. As a starting point, a random block distribution of the raw data over all nodes is assumed. After this, a processing step is done and followed by an intermediate output phase, which writes data back into bricks on the local disk. This is repeated until the high level product is completed. For delivering the product, each *unit* writes the data from its local memory to its local disk. The intermediate products should be saved in the used layout, like *BB*, *NB* or *BN* and is distribution over all participating nodes after each processing step. For further improvement of the *brick locality*, A redundancy brick storing strategy on the system is considered For this, the same *bricks* are placed on different nodes, similar to the *Hadoop File System* [Apa19]. The file space usage increases depending on the redundancy factor.

The symbols used in the remainder of this chapter are described in detail in appendix *Symbols*.

4.1 Parameter Definition

Before starting with detailed description of the optimization model a brief overview over the parameters is given. Figure 4.1 shows on the left side the segmentation of an image in $4 \cdot 4$ data bricks. On the right side of the Figure the corresponding pixel space is shown. The color of the bricks illustrate the owning node. In this example, the blue brick b_{11} lies in the memory of the local node while the orange, green, and violet bricks are stored on remote nodes. Also the data is already initialized to a *BB* pattern with 4 *units* which result in $2 \cdot 2$ blocks in the global memory, each block is mapped to one *unit*. The memory layout borders of each node are shown in blue, orange, green, and violet color. On the right side a zoomed view of the $8 \cdot 8$ center pixels is given. From the perspective of the blue local node, all pixels which are not in its local memory are annotated as remote pixels *rp* and all local pixels with *lp*. The pixels on the borders which are necessary to update the local cells near the border are annotated as *cp* for communication pixel, the whole area is called the *halo* area. In this example a $3 \cdot 3$ stencil is used, which results in a area of 1 pixel around each block, visualized with the yellow margin around the blue block. This *halo* represents the communication overhead for this *unit* for the exemplary stencil operation.

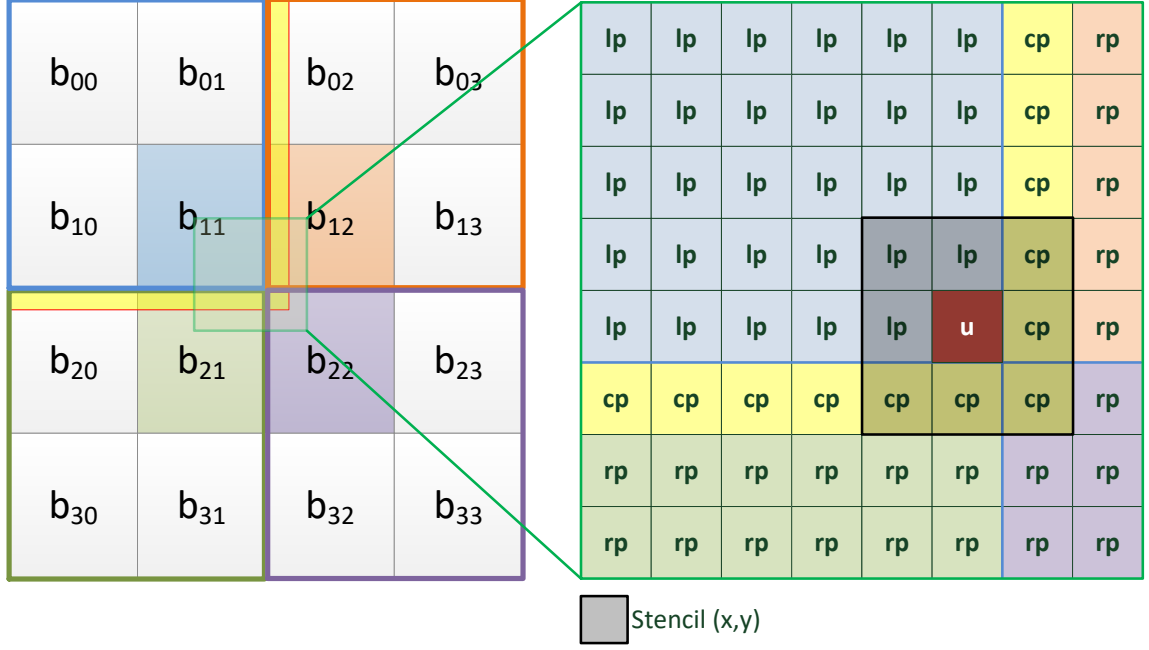


Figure 4.1: Schema of image segmentation into bricks on the left side and example for a single stencil operation on the right side.

4.2 Surface-to-volume and height-to-width Ratio

The communication overhead for the *BB* layout is strongly coupled on the decomposition of the *units* with respect to the extents on the memory, see Figure 4.4. With the *surface-to-volume* ratio the coupling of the dimension in relation to the stencil extents can be expressed.

The *surface-to-volume* ratio is defined as: surface of the block S (surface) divided by the amount of pixel of a block V (volume). S is calculated from the geometric surface of the rectangle of the block while V is calculated from the geometric volume. With an increase of *units* the amount of pixel per *block* (volume) decreases while the necessary communication (surface) increases.

A similar metric is the height-to-width ratio $R_{H/W}$ for stencils with a large extent in only one of the dimensions, e.g. *ST2* or *ST3* is interesting.

The *height-to-width* Ratio $R_{H/W}$ is defined as: height H of the rectangle of the block divided by the width W . Depending on the stencils we would expect lower communication overhead for a small $R_{H/W} \ll 1$ with a large stencil extent in the first dimension and a large $R_{H/W} \gg 1$ with a large stencil extent in the second dimension.

For the given processing problem with a 2D array described in section 3.6 (Figure Fig. 3.3) the *BB*, *BN* and *NB* patterns have the best ratios and should therefore result in a reduced stencil communication overhead, see table 4.1. The stencil communication overhead for *BB* *DASH* pattern is strongly depending on how well the *units* can be factorized in two numbers. In the best case the number of *units* u is square allowing to choose a pattern with \sqrt{u} in each dimension. The worst case is a prime count of *units* which results in a pattern with $1 \cdot u$ blocks, the same as the *NB* pattern. For completeness, Table 4.1 shows also the ratios for

a *TILED(5) TILED(5)* pattern. Contrary to the prior assumption, this pattern generates a $1 : 1$ $R_{S/V}$ and $R_{H/W}$. It will still not be chosen as a fitting pattern because if larger tiles are assumed, the same volume as the *BB* layout is possible, but a more complicate data order in the local *DASH* array will result. Because a *unit* may have multiple tiles in its local array, so the local array cannot be assumed as a $x \cdot y$ large slice of the complete image anymore.

Units	Layout	Surface	Volume	$R_{S/V}$	$R_{H/W}$
8	BB	48	128	$\frac{5}{16}$	$\frac{1}{2}$
8	BN	72	128	$\frac{9}{16}$	$\frac{1}{8}$
8	NB	72	128	$\frac{9}{16}$	$\frac{8}{1}$
8	TILED(5) TILED(5)	16	16	1	1

Table 4.1: $R_{S/V}$ and $R_{H/W}$ for the given examples. [B = BLOCKED, N = NONE]

4.3 Costfunction

For an optimization a cost is necessary, this chapter describes how the cost for the network communication is calculated. At the end, there is a cost value which estimates how many pixels have to be communicated over the network, for switching between two distributions or for loading data from local disk into the *DASH* array.

The cost is calculated on the basis of pixel. As the basis the brick distribution on disk and on the other hand the block distribution in the memory defined by the *DASH* pattern is used. The *brick* distribution is a mapping of all *bricks* with their coordinates to the node where it's stored. In the algorithm this is represented as a simple bitmap, see Figure 4.2. The memory blocks are also represented as a simple bitmap. The memory blocks are depending on the amount of *units* and chosen pattern, as shown in Figures 3.3.

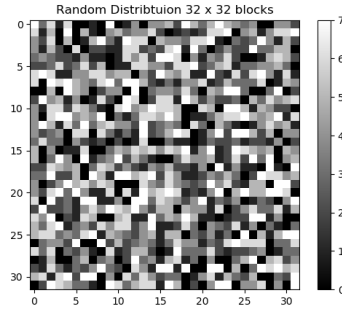


Figure 4.2: Random data distribution on 8 nodes. Each shade of gray represents an assigned node.

For the calculation of the cost the given *brick* distribution d_{start} (e.g. Fig. 4.2) is used as a start point and the difference in the pixel location to the next *block* distribution d_{next} in memory is calculated. Additionally the communication overhead of the stencil *SO* is added.

The cost calculation is based on the *brick* distribution on the disk, and on the *block* distribution in the memory defined by the *DASH* pattern. The *brick* distribution is a

mapping of all bricks with their coordinates to the node where they are stored. In the algorithm this is represented as a bitmap as shown in Figure 4.2. The data distribution represents the memory layout in the *PGAS* depending on the amount of *units* and chosen pattern as depicted in Figure 3.3. This is represented as a bitmap as well. For the calculation of the cost the given data distribution d is used as a starting point and the difference in the pixel location to the next data distribution d_{next} is calculated. Additionally the communication overhead SO is added. The result is a value which represents all pixels which have to be communicated over the network while switching between two distributions.

The remainder of this section describes the cost calculation step by step: At the start, the total amount of the existing pixels in the distribution, called pixel count PC of the given image, is calculated. The image extents in each dimension ext_x and ext_y are derived from the distribution:

$$PC(d) = ext_x \cdot ext_y \quad (4.1)$$

Furthermore the local pixel count $LPC(d, d_{next})$ for the next data distribution d_{next} is defined by the amount of pixels which can be read locally from disk or are in the local memory space of the *DASH unit*. All other pixels have to be transferred over the network or initialized by another *unit*. Thus the LPC is the sum of all pixels where the node location is equal in both distributions:

$$LPC(d, d_{next}) = \sum_{i=0}^{ext_x-1} \sum_{j=0}^{ext_y-1} \begin{cases} 1 & \text{if } d[i][j] = d_{next}[i][j] \\ 0 & \text{else} \end{cases} \quad (4.2)$$

The communication overhead of a layout $SO_{distribution}(d, s)$ calculates the pixels which have to be touched by the algorithm over the network while executing. This part is also affected by the extents in each dimension $sext_x/sext_y$ of the stencils s which are used. The communication overhead depends on the *DASH* pattern as well. It is assumed, that at every *unit* border a network communication will happen. Even though this is not true for *units* on the same node, it simplifies the calculation and should not have a significant impact on the result.

The communication overhead for *BN* and *NB DASH* pattern, $CO_{BN}(d, s)$ and $CO_{NB}(d, s)$, are calculated similarly: The amount of *units* u minus 1 gives the border count, this multiplied by the total image extents (ext_x and ext_y) of the split and the *halo* area results in the total amount of pixel which have to be communicated. The *halo* area is derived from the stencil extents in the corresponding direction. The *halo* extent around each block is $\frac{sext_x-1}{2} / \frac{sext_y-1}{2}$. At each border the communication occurs in both directions, so we have to multiply the result by two resulting in the Equations 4.3 and 4.4:

$$SO_{BN}(d, s) = (u - 1) \cdot ext_x \cdot \left(\frac{sext_x - 1}{2} \cdot 2 \right) \quad (4.3)$$

$$SO_{NB}(d, s) = (u - 1) \cdot ext_y \cdot \left(\frac{sext_y - 1}{2} \cdot 2 \right) \quad (4.4)$$

This was the calculation for the *BN* and *NB*. For the *BB* layout the calculation is different. Like before it is necessary to calculate the amount of borders in the dimension x and y . This is done by first finding the amount of *units* distributed in each dimension. *DASH* is using a prime factorization to split the *units* in each dimension. After having the amount of blocks

in each dimension, the borders can be calculated like above on the other layouts. Now the border count in each dimension $rows/columns$ can be multiplied by the total image extent ext_x/ext_y times the *halo* area. The *halo* has to be calculated like before, resulting in the following Equation 4.5:

$$SO_{BB}(d, s) = (columns - 1) \cdot ext_y \cdot \left(\frac{sext_y - 1}{2} \cdot 2\right) + (rows - 1) \cdot ext_x \cdot \left(\frac{sext_x - 1}{2} \cdot 2\right) \quad (4.5)$$

For the final cost functions the pixel count $PC(d)$ is subtracted from the local pixel count $LPC(d_{next})$ of the given data distribution d_{next} . As a result we have all pixels which have to be transferred over the network during the input phase or all pixels which have to be transferred for switching the data distribution in memory between two processing stages with different stencils.

The communication overhead $SO(d, d_{next}, s)$ depends on the given data distribution d , the next data distribution d_{next} and the stencil s which will be used. At the end, the total cost of the given brick from a given data distribution to a next data distribution with the given stencil can be calculated with the Equation 4.6:

$$C_{trans}(d, d_{next}, s) = PC(d) - LPC(d) + SO(d, d_{next}, s) \quad (4.6)$$

This only holds for transformations from one distribution to another. As described in Section 3.6 fusion of multiple distributions are also possible. For this the different starting distributions d_1, \dots, d_i have to be considered which should be fused in the same next distribution d_{next} . The total cost is calculated with Equation 4.7:

$$C_{fusion}(d_{1..n}, d_{next}, s) = C_{trans}(d_1, d_{next}, s) + \dots + C_{trans}(d_i, d_{next}, s) \quad (4.7)$$

In Figure 4.3 the communication overhead of each *DASH* pattern with increasing amount of *units* is shown with the three different stencils described in Table 5.3. Like mentioned earlier, every time the *units* are a prime number the BB layout behaves like the NB layout, i.e. $SO_{BB}(d, s) = SO_{NB}(d, s)$.

Trough factorization of the numbers of *units* the derived equation becomes volatile. This violate behavior correlate to the $R_{S/V}$, $R_{H/W}$, see Figure 4.4. Therefore, the quantity of possible input *units* is limited in order to reduce the volatile behavior of the equation. If *units* with $R_{H/W} > \frac{1}{4}$ are filtered out, only blocks with a more homogeneous ratio between height and width remain which leads to a better $R_{S/V}$. In other words, the coefficient of variation v is minimized. A low coefficient of variation indicates a less volatile function. The variation without a filter is $v_{nofilter}^2 = 0.96$ whereas the variation using the described filter is reduced to $v_{nofilter}^2 = 0.42$. From this it can be assumed that the best case for the count of *units* is a square number. A square number on a square image will have $R_{H/W} = 1.0$. This should result in the best performance for the *BB* pattern with an identical stencil in x and y . A prime number of *units* should never be chosen, because in this case the *BB* layout will generate a *unit* decomposition with $1 \cdot u$ and this is similar to the *NB* layout. In Figure 4.5 the filtered function for *ST3* is shown.

Figure 4.3 shows that the communication overhead over the network for *ST1* with *BN* pattern and *ST2* with *NB* pattern is 0. The reason for this is that there is no communication in the dimension which contains borders, so the calculation per *unit* is independent from other *units*.

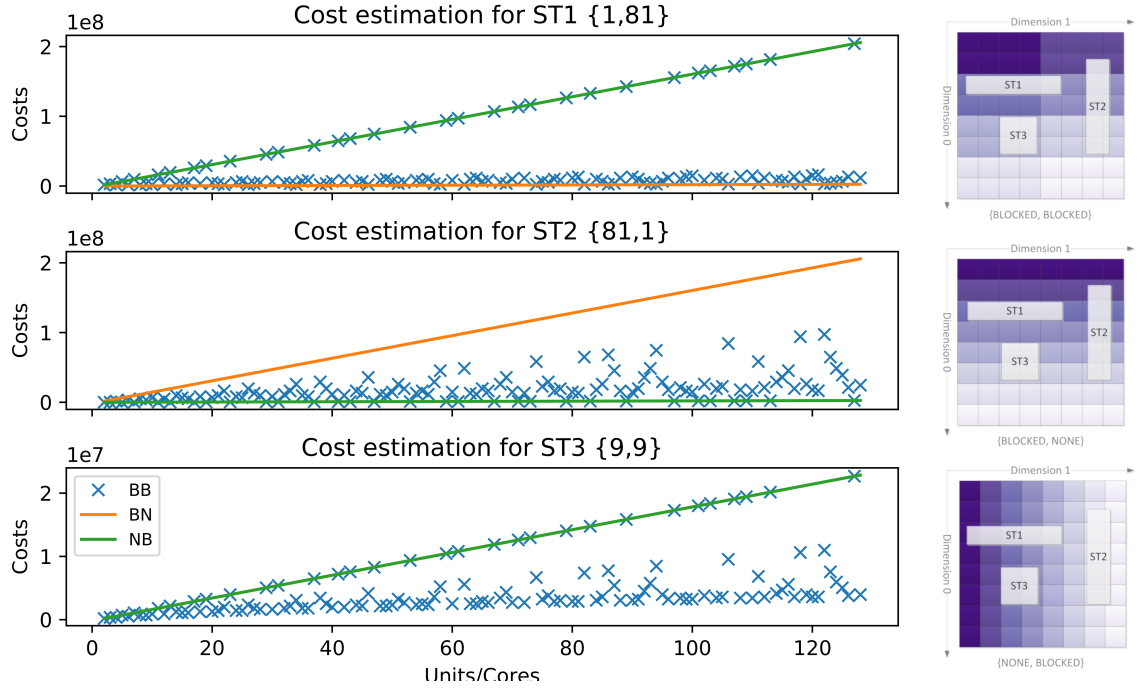


Figure 4.3: $SO(d, s)$ for the different patterns with $ST1$, $ST2$ and $ST3$. Assumed 100% brick locality while initialization. Right side the layout examples with the stencils to show alignment. Bottom plot: $BN = NB$.

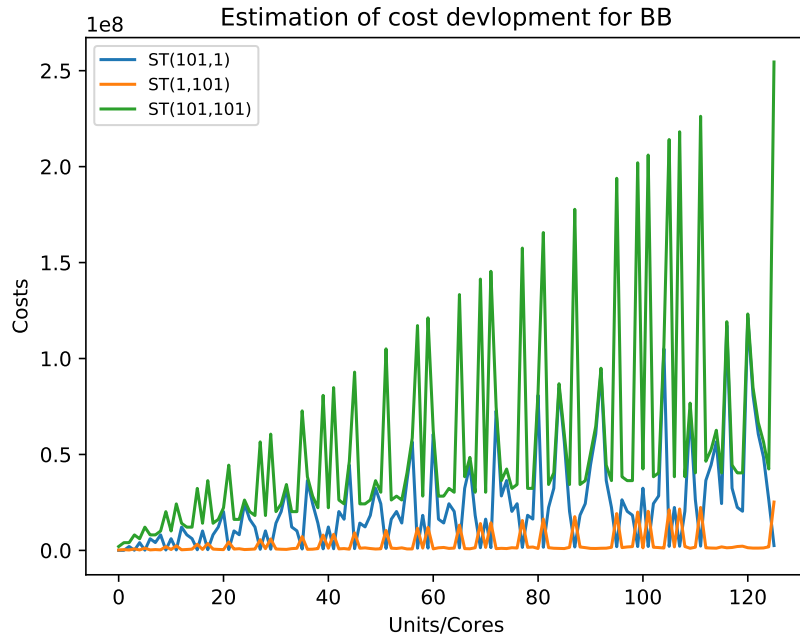


Figure 4.4: $SO(d, s)$ for the BB with different stencil extents.

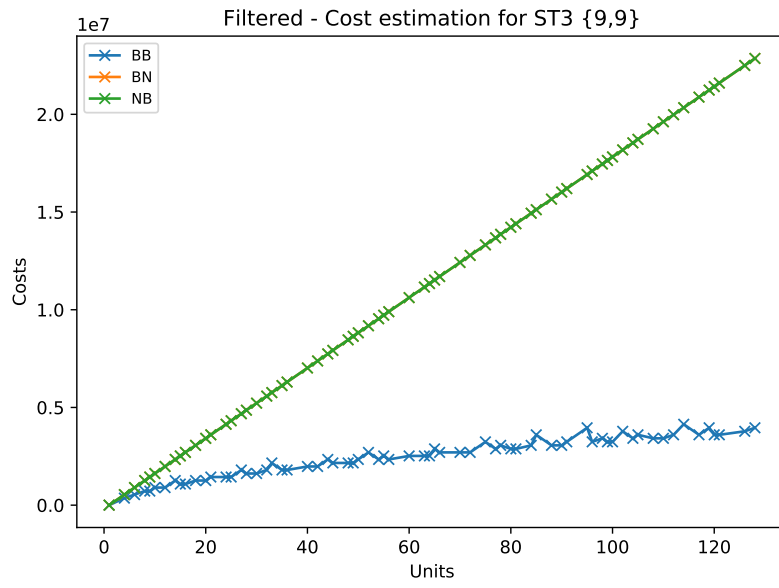


Figure 4.5: Filtered $SO(d, s)$ for the different *units* with $ST3$. Assumed 100% brick locality while initialization. $BN = NB$.

4.4 Optimization Model

A given data distribution d_{opt} is optimal if minimum of pixels is transferred over the network for processing next distribution d_{next} with a given stencil s :

$$d_{opt} = \operatorname{argmin}_{d \in \{d_A, d_B, d_C, d_D, \dots\}} (C(d, d_{next}, s)) \quad (4.8)$$

The next distribution d_{next} could be either of a *BB*, *BN*, or *NB* pattern. The initialization phase should be faster if more local bricks are available, so it should be useful to change the node order or manipulate the *unit* mapping to achieve a higher locality, or use *unit* pinning of the *MPI* framework. But this will result in a large amount of permutations, N it is $N!$ and this already becomes large with a few nodes. For searching the global minimum it is no longer possible to calculate every solution in a reasonable time frame. In this problem space, other strategies have to be used. To reduce the permutations we can try to head for a local minimum. One strategy could be to just switch the first node with another and calculate if the locality is increasing for the patterns. If such a combination is found, we look at the second node and try to find again a combination with locality increase. This is repeated until each node is used for a search or as a change partner. At the end we only have to check $\frac{n(n-1)}{2}$ permutations. But before putting effort into strategy, it is important to know if the effort to find this strategy will pay off. It is important to know how big the impact of inputting none local bricks vs. local bricks on the total performance of all processing steps is. Another possibility for optimization is to vary the amount of *units*, e.g. to gain a better $R_{S/V}$ and $R_{H/W}$ but this will result in a new initialization phase. One advantage of not manipulating the *units* is the ability to keep the data in memory and just change the patterns while processing. Anyway, writing back the intermediate results is a requirement for the *RS* processing to keep intermediate results for reuse in the future. So a new initialization phase could be possible, but only if the effect of a better ratio with less *units* results in a performance gain and this have to be proofed in the experiments.

4.5 Optimization and Worker Software

The following section is a short overview of the implementation of the software framework for testing the optimization concept from the previous Chapter 3. The complete software stack is split in two layers. On the first layer is the *Locality Optimizer for Remote Sensing Data*, a Python program which analyzes the multi-channel *SAR* processing workflow and generates tasks as an output. The second software component is the *Remote Sensing Image Distribute and Processor* and is responsible for executing these tasks and is described in Section 4.5.2.

4.5.1 Locality Optimizer for Remote Sensing Data

For the implementation the Python version 3.7 created with *Anaconda* on an *Ubuntu 18.04.03 LTS* operating system is used. The developed program's task is to find the optimal distribution to speed up the given workflow for multi-channel remote sensing, see Figure 3.4. The input for the *Locality Optimizer for Remote Sensing Data* is a processing graph with multiple channel transformations and channel fusions at the end. Each fusion and transformation defines a stencil for the operation and uses a smoothing filter for workload simulation. After

the optimization of the graph for a data flow with less network communication, it outputs a processing task for each channel transformation and fusion. The resulting processing task consists of operation, input file, and its initial brick distribution, defined as an array as depicted in Figure 4.2. Also the calculated optimal *PGAS DASH* pattern is given as well as the amount of *units* and the participating nodes.

From the initial distribution of each channel and with the cost function, see previous Section 4.3, the lowest cost for the next best distribution for the next step can be calculated for the given task (ST1,ST2,ST3). This will be the start distribution for the next iteration. After all transformations are done for each channel, the fusion steps will be calculated. In the first fusion step the two channels are summed up and in the second fusion step four channels are summed up. Depending on the initial distribution and the applied stencil operations, different channels will have different distributions. Because the fusion process will also apply an operation with a stencil, all two or four different channel distributions have to be aligned in a next distribution.

As a starting point for the distributions of the channels, a random distribution of the raw data bricks on all participating nodes is used. Every brick is stored on a participating node, so it is easier to handle the input data for the experiment. If the knowledge about what the first stencil will be is used, it is also interesting to assume a specific pattern as a starting data distribution. The data distribution is encoded in an array. The value of each element corresponds to the node which stores the data, see Figure 4.2. The master node which optimizes the workflow is responsible for managing the storage information. It is assumed to have every intermediate product stored on the system as a data brick. Each *unit* will write its local data back to disk. Therefore it is possible for the master node to infer on which node the data will be stored after a single processing step.

4.5.2 Remote Sensing Image Distributor and Processor

The second layer is implemented in *C++* and is based on the *DASH-Project* [LM19]. For executing an application which uses the *DASH* framework, the user has to fulfill the requirements for *MPI* on every participating node and the compiled application has to be installed on every node accessible for the user. The user also has to configure a password-less *SSH* login from the master node to every other node. For the *MPI* implementation, the latest *MPICH* release, version 3.3.2 [oBC19], is used. The software is compiled with *GCC 7.4.0* on an *Ubuntu 18.04.03 LTS* operating system.

For the simulation of a real workload, a modified smoothing-algorithm is used. Instead of just using a $3 \cdot 3$ matrix with different weights, it is using a $n \cdot m$ matrix. Depending on the stencil extents, the resulting workload is similar to a focusing algorithm in each dimension of a remote sensing image processing.

$$a_{i,j} = \frac{\sum_{i=-\frac{sext_x}{2}}^m \sum_{j=-\frac{sext_y}{2}}^n a_{i,j}}{m \cdot n} \quad (4.9)$$

The element $a_{i,j}$ at position i, j is the sum of surrounding elements divided by the count of elements $n \cdot m$. Of course, this will not reflect the correct workload of the processing and does not have the same complexity, but it is enough to simulate a load on the CPU. If the original processing algorithms are used, it is possible to optimize the execution time for the cell updates with the knowledge about the algorithm, but this is out of scope of this thesis.

The current processing of the data is done as follows: The *Locality Optimizer for Remote Sensig Data* program determined a list of tasks for each channel. These tasks can be executed in parallel with a resource manager. For the experiment in this thesis, the tasks are executed successively on as many nodes as possible while the execution time is being measured. The execution time is divided in input time, processing time, layout switch time and output time. This allows to decide which processing part benefits the most from optimizations, or to assess the impact of *I/O* or network communication over all processing tasks. Each node is responsible for initializing the data on its own, according to the input distribution. Depending on the *DASH* pattern it could be a local brick or it loads a brick and copies it to the memory of a remote node.

Listing 4.1: Example `mpirun` call of a single processing step wit ST3

```
mpirun -n 64 --hostfile mpi-hr-cluster.txt ~/bin/rsid -l bb \
-p /data/io/input_file calculate -j 9 9 -f smoother
```

There are two different implementations of the smoothing algorithm used in the experiment. The first one (v1) directly operates in the *PGAS* memory space generated with *DASH*, see listing 4.2. In this solution each call to a remote pixel will result in network communication.

Listing 4.2: Implementation v1: Inner loop of smoothing algorithm operating on the *DASH* array

```
for (auto it_x = 0; it_x < stencil_extent_x; it_x++) {
    for (auto it_y = 0; it_y < stencil_extent_y; it_y++)
    {
        Pixel value = dash_data_old[x_start + it_x]
                        [y_start + it_y];
        sum_p += value.p;
    }
}
dash_data_new[x][y].p = sum_p /
    (stencil_extent_x * stencil_extent_y);
```

In contrast to (v1), the next solution (v2) puts more effort into the implementation. It synchronizes the halo regions to a local array before starting with the calculation. In this implementation the calculation for each *unit* is split into three steps: (1) Copying the necessary slice (yellow rectangle in Figure 4.1) to a local array. (2) Calculate every pixel with the stencil operation on the local memory. (3) Copying it back to the local part of the shared memory space of the *DASH* array. The implementation bypasses the repeated calling of remote pixels and uses the *DASH* array only for data exchange and does not work directly in the global shared memory space. The halo sizes are calculated by the stencil size in each dimension. The resulting *local_copy* is the local array of the *unit* plus a halo in each dimension. The developers of the *DASH* framework are planning to release a similar feature in the next release. They will go even further with the implementation and allow the developer to first calculate on the local part and after updating the halo in the background operate in the halo area. In this implementation the complete array plus halo is sliced line wise from the global array. For each *unit* which holds a part of the image border, a case distinction has to be made. Depending on the border side (top, down, left, right) it should not copy a halo which lies outside of the *DASH* array. The Listing 4.3 shows the function

calls on all three steps. Inside the *copy_dash_to_local* functions the *dash::copy* function is used to move data, line wise, from the global shared memory to a local buffer. The call has to be line wise, because the function is implemented in way to slice out one consecutive memory line of the array from the *PGAS* memory space to the local memory buffer. A function to copy a 2D slice will follow in the future releases of *DASH*. Instead the *copy_local_to_dash* just copies the whole *result_copy* array to the local *DASH* array part of the *unit* in one large *dash::copy* call.

Listing 4.3: Implementation v2: Copy, calculate and copy back function calls with previously generated halo extents in each direction.

```

auto tmp_ext_x = local_extent_x + halo_n + halo_s;
auto tmp_ext_y = local_extent_y + halo_e + halo_w;
Pixel *local_copy = new Pixel[tmp_ext_x * tmp_ext_y];
Pixel *result_copy = new Pixel[local_extent_x * local_extent_y];

copy_dash_to_local(dash_array, local_copy,
                  ul_x, ul_y, lr_x, lr_y,
                  halo_n, halo_e, halo_s, halo_w);

calculate_with_stencil(local_copy, result_copy,
                      ul_x, ul_y, lr_x, lr_y,
                      halo_n, halo_e, halo_s, halo_w);
dash::barrier();
copy_local_to_dash(dash_array, result_copy, ul_x, ul_y);

```


5 Results & Discussion

5.1 System Description

The evaluation experiments are executed using both a heterogeneous hardware environment at DLR and a homogeneous *HPC* system at the *Leibniz-Rechenzentrum* in *Munich* [Rec19].

The DLR system consists of four processing servers with the name *HR-SLX005/HR-SLX007/HR-TEST001/HR-SLX012*. The respective hardware specification is shown in the Table 5.1. *HR-TEST001* is a virtual server. The virtualization of this host will impact the read/write performance to the disk, because some VM's can use the same disk space on a raid system. To limit this undesired behavior, *HR-TEST001* is the only virtual server running on the VM-Host with exclusive access to all resources of the host. Nevertheless, to avoid unintentional impact on the evaluation results, this server will not be used for read/write performance testing. Figure 5.1 shows an overview of the infrastructure. A *GPFS* system is shown on the left with three *I/O* servers which are connected by a 10 Gb Ethernet connection to the four processing servers. The 10 Gb Ethernet is only used for file transfer and communication between the processing servers. The *Network File System* (NFS) service also running on the file servers (*HR-FS02/HR-FS04/HR-FS05*). On the right side the hardware stack for *HR-TEST001* and its VM-Host is depicted. Each of the processing servers is accessible directly via SSH and accessible without any login nodes.

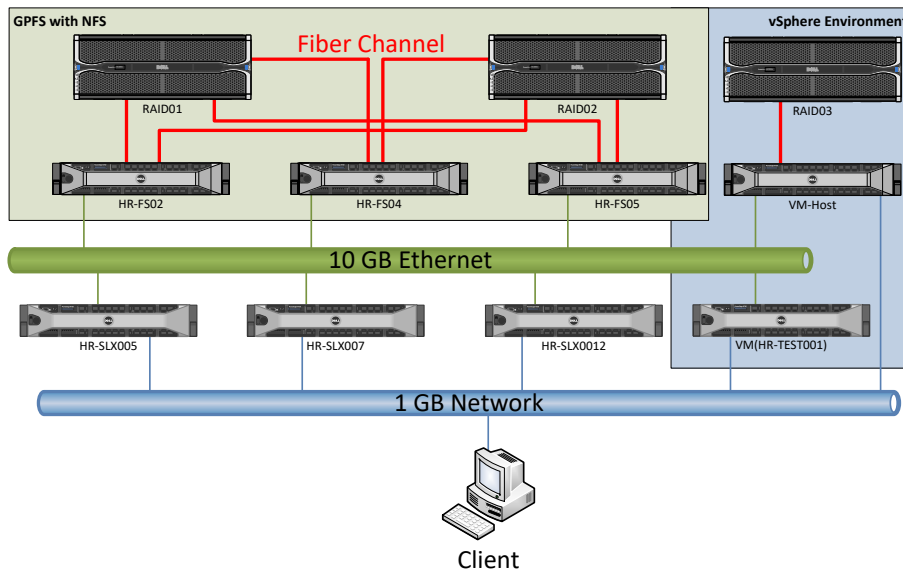


Figure 5.1: *HR-Cluster* Hardware overview

Name	HR-SLX005	HR-SLX007	HR-TEST001	HR-SLX0012
Processor	Xeon E5-2690	Xeon Gold 6130	Xeon Gold 5120	Xeon E5-2698v4
Frequency	2.6 GHz	2.1 GHz	2.2 GHz	2.2GHz
Cores per Node	24	32	28	40
Hyperthreading	Yes	Yes	Yes	Yes
Cache L3	30 MB	22 MB	19.25 MB	50 MB
Memory	512 GB	768 GB	256 GB	512 GB
Filesystems	SSD	SSD	SSD	SSD
Interconnection	10 Gb	10 Gb	10 Gb	10 Gb

Table 5.1: Server in the DLR hardware configuration overview

The *operating system* (OS) Ubuntu 18.04.2 *LTS* is used for processing servers. The *boost* library is installed in version 1.66 as well as the latest *MPICH* version 3.3.1 and [oBC19] as the *MPI* basis framework. The *DASH-Project* framework is installed with the latest development version 0.4.0 from Jan. 9, 2019[LM19].

The *SuperMUC-NG* is a homogeneous *HPC* system of the *Leibniz-Rechenzentrum* in Munich. An overview of the hardware configuration based on [Rec19] is given in Table 5.2.

Name	Thin Nodes	Fat Nodes
Processor	Xeon Platinum 8174	Xeon Platinum 8174
Node Count	6,366	144
Frequency	3.10 GHz	3.10 GHz
Cores per Node	48	48
Hyperthreading	Yes	Yes
Cache L3	33 MB	33 MB
Memory per Node	96 GB	768GB
Filesystems	GPFS	GPFS
Interconnection	Fat Tree	Fat Tree

Table 5.2: *SuperMUC-NG* hardware configuration overview

All nodes on the *Leibniz-Rechenzentrum* are running the OS *SUSE Linux Enterprise Server 12 SP3*. For the test execution on the *SuperMUC-NG*, the *boost* library is installed in version 1.66 and as the *MPI* basis framework the *Intel MPI* library version 2019 update 6 build 20191024 is used. The *DASH-Project* framework is installed with the latest development version 0.4.0 from Jan. 9, 2019 [LM19].

5.2 Performance Evaluation

5.2.1 Read/Write Speed

The read/write speed is defined by the time $T(B)$ a processor takes to load a single data brick B of the size $brext_x * brext_y$ pixels. This is an important metric to evaluate if we can achieve a better performance on multiple nodes with a local file system or a shared network file system. The network file system is expected to become the input speed bottleneck for high data throughput processing allowing the multi node solution to perform better.

5.2.2 Speedup

The speedup $S_p(n)$ is defined following Amdahl's Law:

The performance improvements to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.
[Amd67]

This law provides a generic limit of maximal speedup of any parallel computation and is paraphrased as follows:

$$Speedup = \frac{1}{r_s + \frac{r_p}{n}} \quad (5.1)$$

where $r_s + r_p = 1$ and r_s represents the ratio of the sequential portion and r_p the parallel portion of one program [Amd67]. From this, the definition for speedup $S(p_1, p_m, n)$ is derived as the ratio of the time required by a sequential application $T(p_1, n)$ to a parallel version $T(p_m, n)$:

$$S(p_1, p_m, n) = \frac{T(p_1, n)}{T(p_m, n)} \quad (5.2)$$

where p_m is the number of processors used to process input of size n . In the best case, $S(p_1, p_m, n)$ is near equal to one, but does not reach one. The reason is the communication overhead between parallel and sequential portion r_s of the application. This could be also used to compare execution times from different amounts of processors p .

5.2.3 Efficiency

The efficiency $E_p(n)$ for a problem of size n is defined as the ratio of the time required by an application using one processor to the time required by a parallel application using p processors multiplied by the value p . In the theory $E_p(n)$ could be equal to 1, but as described before we cannot attain this optimum because of communication overhead and the sequential portion r_s of the application.

$$E_p(n) = \frac{T_1(n)}{p \cdot T_p(n)} \quad (5.3)$$

5.2.4 Strong and Weak Scaling

With a scalability test it is possible to measure the ability of an application with varying problem sizes or numbers of processors. In general its possible to distinguish between *strong scaling* and *weak scaling*.

For *strong scaling* the problem size is fixed and the number of processors, which take part in the computation, is increased. This results in a reduced workload per processor, mostly used for long-running CPU-bound applications to find a setup with a reasonable execution time and which uses a moderate amount of resources. *Strong scaling* is normally affected by *Amdahl's Law*, in order to be precise, affected by the increasing communication overhead in comparison to the calculation time. In the *RS* image processing a long-running CPU-bound application is the case for a really large stencil computation or if we apply many stencils in a row in one processing step.

For *weak scaling* the problem size is fixed per processor, so increasing the number of processors increases the computed problem size equally. The result is a constant workload per processor. *Weak scaling* is mostly interesting for large memory-bound applications where a distribution to multi nodes is necessary trough memory limitations on a single node which is expected to be critical for the upcoming *RS* image trends with growing image sizes this is true for our application. Following [MML17] it is mostly interesting for algorithms with time complexity $O(n)$. It is derived from a tending upwards time that the overhead due to parallelisms is also increasing or that the algorithm is not truly $O(N)$.

For the applications investigated in this thesis, both aspects are interesting because a CPU-bound application with a long runtime is investigated which uses large stencils on problem sizes that do not fit onto a single node.

5.3 Experiments

The experimenting started with the evaluation of the overall read/write performance of the system described in Section 5.1. For this purpose, a disk load test was performed. After that the impact of the brick locality is evaluated. Following this experiment a test for strong and weak scaling is conducted. In the next steps the computing speed and impact of each stencil to the different *DASH* patterns on two different implementations is evaluated in the try to find weighting parameters for the cost function. The experimenting phase is concluded by a test of the optimization model on a complete processing chain and the discussion of the results.

5.3.1 Brick Locality

Splitting data up in bricks and storing them locally while adding a redundancy is increasing the overall read/write performance of the system with minor investment in the data storage back-end because only already existing resources are utilized. Therefore a load testing based evaluation of different file system structures is done in order to determine the maximum read/write speed achievable in the given environment of this work. The differences in speed between local SSD and the *Network File System* (NFS) are compared. A test with the *GPFS* system on the *HR-Cluster* would give the chance to compare it with the large scale *GPFS* system on the *SuperMUC-NG*. But the test was not executed on *GPFS* on the *HR-Cluster* because the performance was already low with a single node, for further analysis the problem

with the *GPFS* has to be fixed first. For the evaluation experiment, the *Linux* application *dd* was used. In order to achieve comparable results, all caches were deleted locally while data was read or written on the server. Because the RAID System lacked the possibility to delete the caches, the retrieved results for repeated reading outperformed the comparison group. In the test 4 GB data bricks B , which represents a $65,636 * 65,636$ pixel image on each node, were used to generate a representative *I/O* load. The test shows $T_{read}(B) = 28.51s$ and $T_{write}(B) = 96.44s$ with 4 nodes in parallel on the *NFS*. In comparison to that, local data handling took in average $T_{read}(B) = 11.22s$ and $T_{write}(B) = 24.03s$ in average on each node.

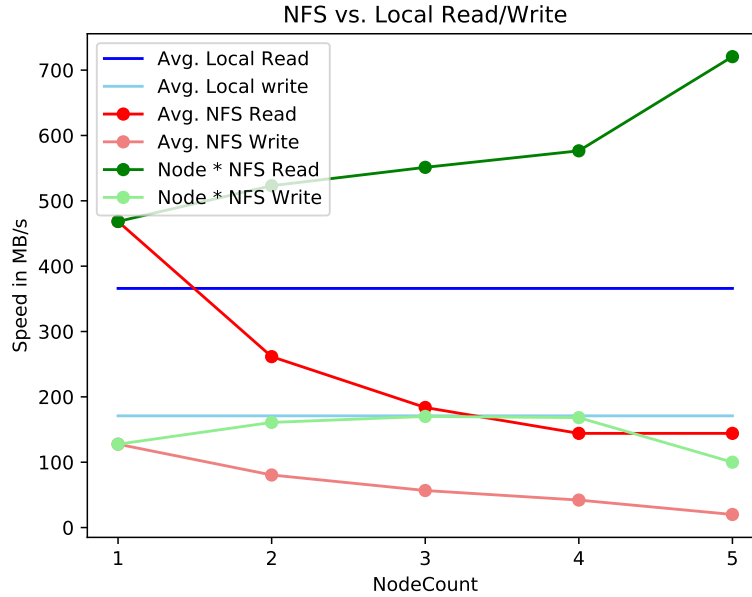


Figure 5.2: Comparison of read/write speed of local bricks with *NFS* on multi nodes. Green lines are summed average read/write performance of the single nodes.

From the experiment with *NFS* it is estimated that the network storage infrastructure will reach its interconnection limit between nodes and network storage already with a few nodes reading or writing in parallel as depicted in Figure 5.2. The maximum read speed of the used storage infrastructure is around 550 MB/s and the maximum write speed is around 220 MB/s depending on the access pattern and how good the cache on the RAID system can be utilized. In comparison to the *SuperMUC-NG*, which is able to deliver 500 GB/s with a *GPFS* file system, this is a quite low performance. But if all 6,336 nodes on the *SuperMUC-NG* would read/write in parallel this will also lead to a small speed per node with around 80 MB/s.

Not only the general loading from a local disk compared to a *NFS* is important, it is also necessary to know how large the difference in the initialization speed between a 100% brick locality and 0% brick locality is and on which part it takes the most time. Therefore both the read time of the image from disk/*NFS* as well as the copy to the *DASH* array time are examined. For the initialization phase the best performance should be achieved if 100% brick locality is reached, which means each *unit* is able to load the brick from the

local disk so that no network traffic will be generated. For this assumption an evaluation experiment on 3 nodes with 64 *units* was conducted with an image of $131,072 * 131,072$ pixels which were segmented in 64 quadratic bricks. This experiment was executed directly with the developed *RSIDP* software. Before each run the local disk caches were cleared on each server. In this locality test it is assumed that each *unit* can read one block and no *unit* has to read two blocks. If this would not be the case, the initialization time would be at least doubled, because some *units* have to read two blocks and other *units* need to wait for them at the synchronization point. The measured reference time for the input of all bricks to the memory for 100% locality is $T_{input}(I) = 19.47s$ and writing took $T_{output}(I) = 36.43s$. For comparison, the *NFS* with the same test was slower and took $T_{input}(I) = 30.56s$ for the input and $T_{output}(I) = 57.9s$ for the output. With 50% locality, which means only the half of the *units* could be locally read, half of the data is initialized by a remote unit. The evaluation experiment determined $T_{input}(I) = 25.91s$ and $T_{output}(I) = 36.83s$. In the worst case, no block is stored locally and all blocks have to be loaded from a remote unit, so we have an even higher network traffic for the initialization. In this case it took $T_{input}(I) = 26.58$ and $T_{output}(I) = 37.42s$. T_{output} is similar in all but the *NFS* experiments, because it is independent from the initial brick locality, it is always writing back on the local node so that a 100% locality is achieved.

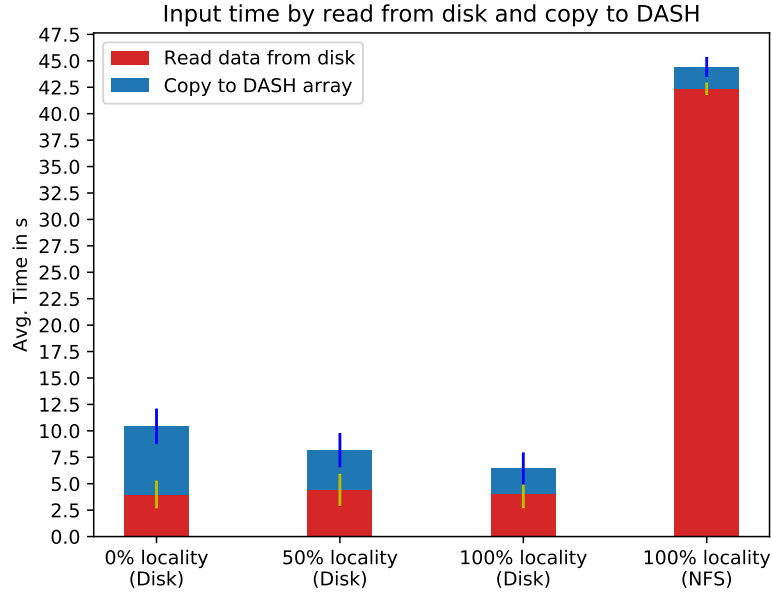


Figure 5.3: Average Initialization time on 3 nodes on the *HR-Cluster* in seconds by read image and copy to *DASH* with different brick localities, standard deviation displayed on top.

The Figure 5.3 shows the average time per *unit* to read the data from the disk and copy them to the *DASH* array. It can be seen that with decreasing brick locality the average copy time to the *DASH* array is increasing. On 100% brick locality it took in average 5s to read and copy the data to the remote *units* while it took 2.5s to read the data from disk. On 0% brick locality it took in average 7s to read and copy the data to the remote *units*

while it took 3s to read the data from disk. On the central *NFS* system with 100% locality it took an average of 45s to copy and read. The experiment shows a best case scenario with 100% locality which lead to an average speedup of factor 6 compared to reading the data from the central *NFS* file share. Also the 0% brick locality shows nearly a average speedup of 4 in comparison to *NFS*. The reason is the local loading speed dominating the speed of the *RAID* system behind the *NFS* server and the network utilization, which is used in both cases, is less important at this point. Between 0%locality and 100% locality a improvement about 25% was achieved. Extrapolated to larger image size up to 1 *TB* this could bring a strong improvement in the initialization phase.

5.3.2 Locality in Multi-Channel RS Image Processing with Potentially Large Stencils

In this experiment the whole processing chain from Section 3.6 is executed. The motivation behind this experiment is to check if the optimization function gives valid decisions about the memory layout changes. As a starting point, an image will be segmented in bricks and be distributed in different distributions to all participating nodes. As task the workflow described in Figure 3.4 is used. The workflow starts with 8 channels of raw data and generates 6 high level products from it. An artificial gray image with is used as raw data set for each channel. For the test 64 *units* are used on 3 nodes.

A small image size of 8000 * 8000 pixels was used, compared to the future products of about 1TB large images. This experiment will not proof that *brick locality* improvements result in better execution times because of the small problem size. This limiting factor was necessary in order to achieve execution times which allowed to make incremental experiments without too long experiment duration. But from this small experiment we can proof if the optimization function result in reasonable decisions.

Figure 5.4 shows the optimization result from the *Locality Optimizer for Remote Sensing Data* application for 3 nodes and 64 *units* with a random distribution for each channel and the stencil operations from Table 5.3. The stencils are chosen to be similar in shape to current *RS* image processing stencils used on real *SAR* data. The first stencil *ST1* is aligned to a *BN* while *ST2* is aligned to a *NB* layout and *ST3* is aligned to a *BB* layout. Additional *ST1* should benefit at most from the cache on the nodes, because of the default *Row Major* order in a *DASH* array [FF16] and should show the best performance in total throughput. It is assumed that the performance of each stencil can be improved by choosing a layout which reduces the network communication which equals to increasing the data locality. Underneath the arrows in Figure 5.4 the estimated execution times from Table 5.4 and the cost for the layout switch with communication is depicted. The execution time was determined by the average of 10 executions for the specific stencil layout combination without considering *I/O*. It can be seen that the first layout, which generates the least cost, is propagated through the whole processing chain. With relative small stencils the layout switching cost plus communication costs are always higher than working with the current layout and the high communication cost of the stencil. But looking at the execution times of the proposed processing chain, it can be seen that in the second step *ST2* with the layout *BN* it rises by the factor 72 compared to the previous step *ST1* on the first 7 channels. Even the last channel, which performs best for *ST2* with 103.45s, cannot reach the best performance with layout *NB* of 9.99s. So for *ST2*, the worst performing layouts were chosen. At this point a weight should be introduce to treat *brick locality* different from the stencil communication costs *SO*, but to find such a weight more performance test are necessary for different layout units combinations, see next Section 5.3.3.

But to check if the optimization function gives reasonable decisions in general the *brick locality*, *LPC* in Equation 4.7, is removed and calculation are only with *block locality* cost.

In this experiment only the communication cost are considered for the memory layout decision, the result differs form the previously presented experiment outcome. Figure 5.5 shows the optimization result for these assumptions from the *Locality Optimizer for Remote Sensing Data* application for 3 nodes and 64 *units* with a random distribution for each channel and the stencil operations from Table 1.1. Underneath the arrows in Figure 5.4 the estimated executions times from the Table 5.4 and the result of cost function are shown

Name	Dimension 0	Dimension 1	Aligned Pattern
<i>ST1</i>	1	81	BN
<i>ST2</i>	81	1	NB
<i>ST3</i>	9	9	BB

Table 5.3: Stencil Set 1: used in divers experiments as basis stencils. [N = None, B = Blocked]

Layout	Stencil x	Stencil y	Time/s	Input/s	Output/s
BB	1	81	104.3	0.12	0.04
BN	1	81	9.95	0.05	0.04
NB	1	81	731.5	0.13	0.09
BB	81	1	103.45	0.02	0.09
BN	81	1	762.17	0.12	0.06
NB	81	1	9.99	0.03	0.05
BB	9	9	11.69	0.02	0.06
BN	9	9	97.93	0.12	0.06
NB	9	9	94.01	0.27	0.08s

Table 5.4: Average execution times of 10 tests for the different stencils/layout combinations with implementation v1 on a 8000 * 8000 pixel image in the *HR-Cluster*.

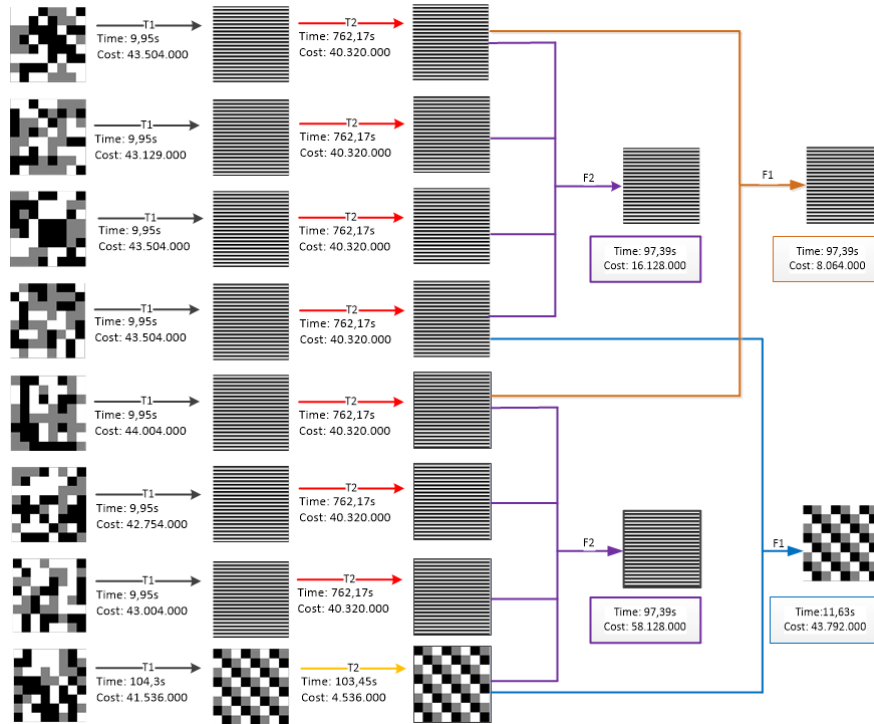


Figure 5.4: Example optimization result for 64 bricks distributed to 3 nodes and processed with 64 units. Each gray tone represents data located on the specific node.

according to the previous results.

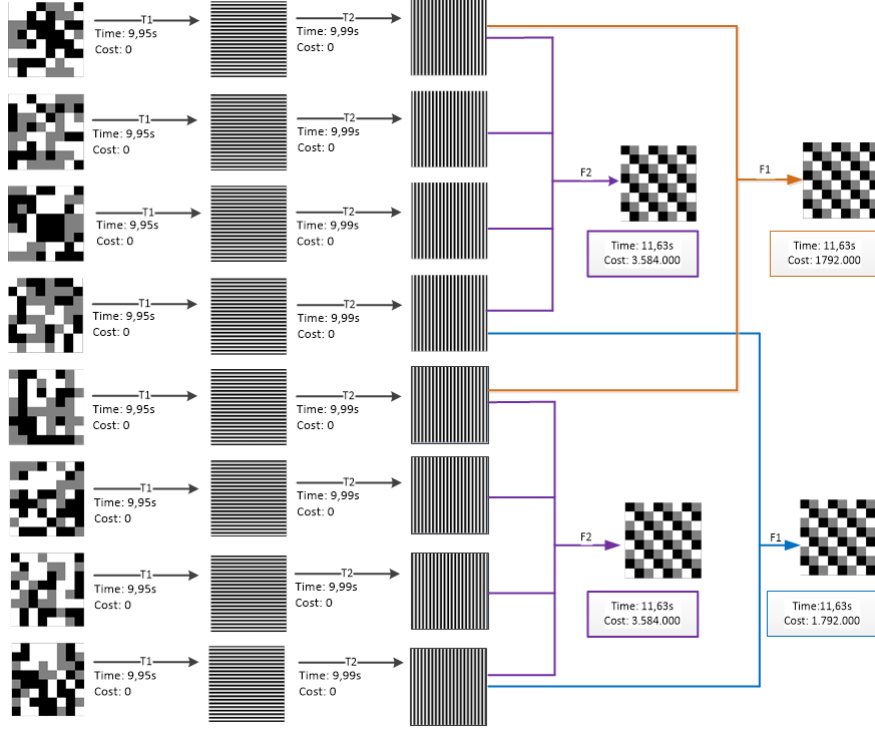


Figure 5.5: Example optimization result for 64 bricks distributed to 3 nodes and processed with 64 *units* without considering brick locality. Each gray tone represents data located on the specific node.

The optimizer now chooses the layout which generates less communication cost and does not consider the initialization phase or layout switches. The random distribution at the beginning does not matter now. The first two steps *ST1* and *ST2* will generate 0 costs because the stencil only has one dimension and with the given layout there are no borders in this dimension and so no costs at all. So only the array with the *BB DASH* layout generates communication costs. With this approach, more practical results are achieved as shown in Table 5.4. But it can be seen that the results show the same behavior as in the previous experiment if the stencils are increased in each dimension so that the stencil communication costs starts to dominate the *brick locality* costs, see new stencils in Table 5.5 and results in Figure 5.6. These results show that the proposed optimization model works as intended: the layout switching cost plus low communication cost of a better fitting layout are lower than staying in the same layout, which gives 0 costs, plus the higher communication cost of a bad fitting layout.

For all tests, the same initial distribution was used but in reality the system is supposed to have different distributions as data sets. Also, there was the idea to have the block redundant on the nodes to have an increased locality. This will also only move the point where it's worth to switch the layout and the optimization function will tend to stay at the current layout instead of switching to the better performing layout. In both cases, with redundancy or considering the normal *brick locality* loading Before further experiments with the optimization function can be carried out, the communication effort of the stencils must

Name	Dimension 0	Dimension 1	Aligned Pattern
$ST1$	1	101	BN
$ST2$	101	1	NB
$ST3$	128	128	BB

Table 5.5: Stencil Set 2: larger stencils in comparison to set 1 to increase the stencils communication overhead SO . [N = None, B = Blocked]

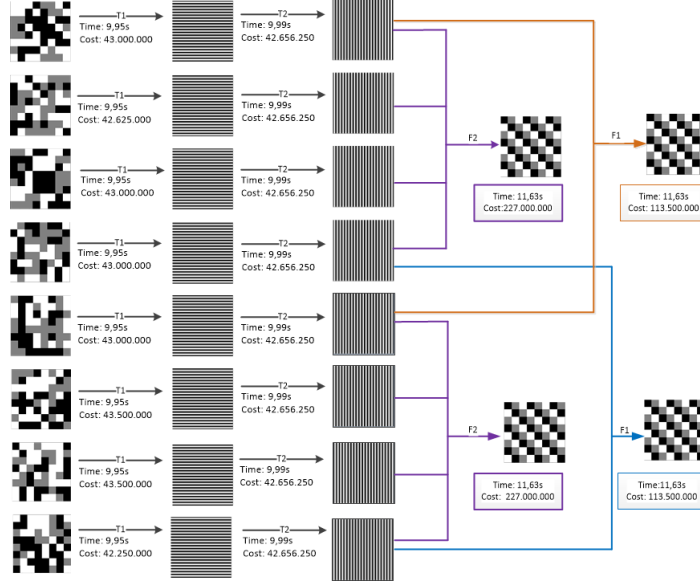


Figure 5.6: Example optimization result for 64 bricks distributed to 3 nodes and processed with 64 *units* without considering brick locality using large stencils. Each gray tone represents data located on the specific node.

be weighted against the actual block loading times. We can conclude the layouts are choose correctly based on the cost function but a weight is necessary to treat *brick locality* different from *block locality*.

5.3.3 Layout Performance with Different Stencils

The previous experiment shows the optimization function worked as intended, but weights are necessary to treat *brick locality* and *block locality* in the context of the total execution time estimation correctly. The hypothesis is, if a large stencil is used which is not aligned to the *DASH* memory pattern, the overall performance decreases because of the increased communication overhead. It could be possible that a communication in dimension 0 is more important than in dimension 1. To test the impact of the dimension extents in this experiments three stencils with the same workload but different extents, in each dimension are used as listed in previous Section 5.3.1 Table 5.3. The prediction for the layouts with the stencils is: *BB* should result in the best performance for a stencil with equal extents in each dimension, *NB* should result in the best performance for the largest extent on the first dimension, and *BN* should outperform the other layouts for stencils with the largest extent on the second dimension

For this test we create a *DASH* array of the size $20.000 * 20.000 * 3 \text{ Byte} = 382 \text{ MB}$ and apply a smoothing filter for simulating a workload. During the simulation, the data is generated on the fly so no *I/O* happened on the disk in these experiments. As a test environment the two nodes *HR-SLX005*, and *HR-SLX007* of the *HR-Cluster* are used which are connected with a 10 Gb Ethernet connection. No changes in the *unit* pinning were made, so the *units* are distributed to the nodes in a round robin manner.

The first test was executed with implementation (v1) which operates directly on the *DASH* distributed array. The tests were executed 5 times and the average execution time was used for the overall performance evaluation. The results of how well the different stencils operate on the layouts *BB*, *NB* and *BN* is shown in the Figure 5.7. Also, the confidence interval is displayed which shows the minimum and maximum execution time of an iteration.

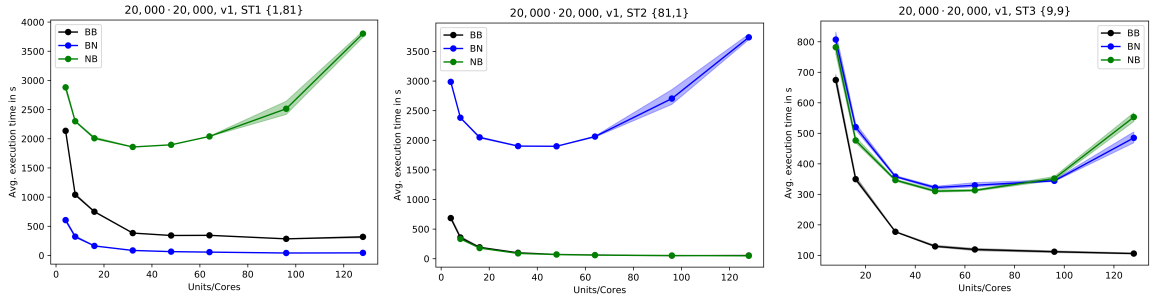


Figure 5.7: Average execution time in seconds of *ST1*, *ST2* and *ST3* with 2 nodes with implementation v1 on the *HR-Cluster*.

The *ST1* with *DASH* layout *BN* produces a throughput higher by the factor of 5 in comparison to *NB*. About 32 cores the plot start to increase with the layout *BB* and *NB* instead of decreasing further. At this point a large network saturation was observed on the *HR-Cluster* system, which could be the reason for a increasing execution time. But we can conclude for this implementation each layout delivers the best performance on the predicted layouts. *ST1* is the fastest on *BN*, *ST2* performed well on the predicted layout *NB* and also *ST3* is the fastest on the *BB* layout. Due to performance issue it is not possible to derive any weights here.

Interesting is the result from *ST2* with the *BB* layout. It produces the similar performance

to the suggested optimal layout *NB*. At this point the regularity of the *unit* distribution becomes a problem. In these experiments just two nodes and different counts of *units* (1, 2, 4, 8, 16, 32, 48, 64) were used. With this amount of units, a segmentation with an even number of *units* in each dimension was created for each distribution higher than 2. The *MPI* default *unit* distribution uses round robin for *unit* to node pinning. In a result the *BB* layout appears identical to the *NB* layout when focusing on the node numbers instead of the *unit* numbers. In Figure 5.8 both layouts with 8 *units* on 2 nodes are shown. The network communication happens only on the block borders in dimension 1 while the communication on dimension 0 happens on the node itself. It can be seen, that the network communication with a *NB* layout is increased due to the node-separating behavior of the blocks. Additionally, the experiment was affected by the restriction of *units* with a overall good $R_{H/W}$ ratio. For a improved test, also numbers of *units* which are more uneven to factorize into two numbers should be considered.

Also, *ST3* with *BB* layout should benefit from a better *surface-to-volume* ratio (3 : 2) but has to communicate with other *units* on the same node. In comparison *NB* layout has no inter process communication at all, but the *surface-to-volume* ratio is worse (18 : 8). This is also the reason for *ST1* with *BB* to have a better performance then *BN* but instead of fast inter node communication the algorithm has to communicate over the network at the read line, see Figure 5.8.

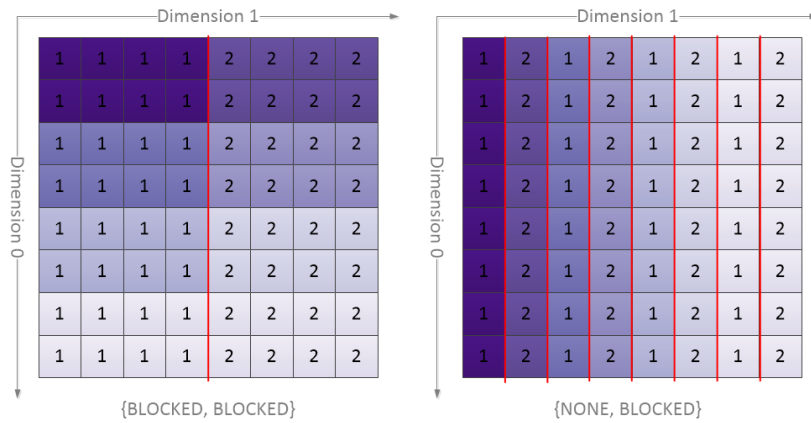


Figure 5.8: Comparison of *BB* and *NB* pattern distribution on two nodes with eight *units*. Numbers display the node assignment of each brick. The block colors show the *unit*. The red lines mark borders where network communication is necessary.

To test the more uneven distribution, the experiment is expanded for *ST2* to three nodes and to also using block segmentation with odd numbers, not dividable by three, to have also a network communication in the first dimension while using *BB* layout. In addition, prime numbers of *units* are discarded. The result is shown in Figure 5.9. The *BB* layout is now much slower compared to the *NB* layout. But now a new effect is visible: If a square number for *units* is used, the overall performance is slightly better. This could be an effect of the overall better $R_{H/V}$ ratio with square numbers of *units* [4, 16, 25, 64, 100, 121], shown in the first plot of Figure 5.9. E.g. for 124 *units* the $R_{H/W}$ ratio is 7.74 resulting in an execution time of 608.05s. The nearest square number of *units* is 121 which has a $R_{H/W}$ ratio of 1 and results in an execution time of 254.61s. This is a performance increase by the factor of 2.39 even though there is less computing power available. This behavior is

correlating with the prediction of the cost function, shown in Figure 4.3.

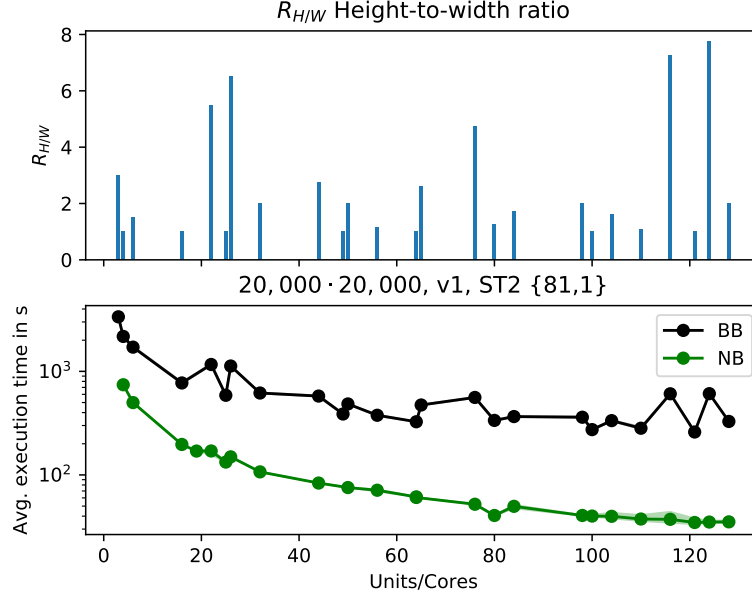


Figure 5.9: $R_{S/V}$ ratio compare to the execution time in seconds of ST2 with 3 nodes for the patterns BB and NB.

The fastest execution time achieved for 128 *units* is 43.1s by *ST1*, 45.46s by *ST2*, and 104.34s by *ST3*. The cache could be a reason for the overall good performance of *ST1* because *DASH* arrays are stored by default in *Row Major* order [FF16]. *ST1* can benefit from data already loaded into the cache lines, while *ST2* produces more cache misses, because every next value will be in another cache line and this line could be missing in the cache. *ST3* could benefit slightly from cache but will also generate more communication overhead in each dimension. If the cache miss rates are really high it could be useful to invest in code optimization for better cache usage, to be able to compare the layout performance better. To check the assumption of performance issues is correlated with cache miss rates, the program was executed again with the *perf* kernel tool of *Linux* to monitor cache misses per *unit* while executing on the nodes *HR-SLX005* and *HR-SLX012*. The results are listed in Table 5.6. The test was executed with 42 *units* and 120 *units* with the best and worst fitting layout for the given stencils, derived from the performance test before. Furthermore, if a correlation between the cache miss rate and the performance gap per stencil between the different layouts would be observed, then the communication overhead should not be the focus for performance optimization. Table 5.6 shows the average cache misses in percent over all *units* including the respective standard deviations. The difference in the cache miss rates between the different stencils is small and does not correlate with the huge performance gap between the different stencils on different layouts. The huge standard deviation over all test is explained by the different cache size of both nodes. Looking at the difference between the node *HR-SLX005* and the *HR-SLX012*, there are less cache misses produced on *HR-SLX012* because of the larger cache with 50 MB in comparison to the 30 MB of *HR-SLX005*, see Table 5.1. The standard deviation of *ST1* with *BN* and *ST2* with *NB* is

much lower compared to the the other tests, because of the cache miss rate was increasing on *HR-SLX0012* and decreasing on *HR-SLX005*. The average cache miss rates were still similar to the other tests.

Speed	Stencil	Layout	cache-misses 42 units	standard deviation	cache-misses 120 units	standard deviation
Fast	<i>ST1</i>	BN	26.10%	7.79%	27.95%	8.35%
Slow	<i>ST1</i>	NB	30.30%	28.06%	31.79%	22.69%
Fast	<i>ST2</i>	NB	28.30%	24.32%	23.95%	7.27%
Slow	<i>ST2</i>	BN	30.68%	28.68%	28.40%	20.16%
Fast	<i>ST3</i>	BB	20.80%	18.91%	30.87%	20.03%
Slow	<i>ST3</i>	BN	32.75%	21.08%	28.90%	18.95%

Table 5.6: Average cache-misses and standard deviation captured with *perf* with, 20.000 * 20.000 pixel image, with implementation v1. [N = None, B = Blocked]

Because of the overall poor performance of implementation v1, a second implementation v2 was implemented, see Section 4.5.2 for implementation details and difference to v1. Looking at the performance of the second implementation (v2), the result looks quite different. The following tests were executed 10 times allowing to use the average execution time as overall performance metric. The data size is the same as before, a 20.000 * 20.000 pixel array. The result is displayed in the Figure 5.10. Also, the confidence intervals are displayed showing the minimum and maximum execution times.

The experiment with *ST1*, seen in Figure 5.10 first plot, shows that the *BN* layout is the fastest, similar to the experiment on implementation (v1) before. *NB* performs again the worst because of the large communication overhead. From 2 to 32 units a strong decrease in execution time is observable with every layout. For *NB* with 2 units the execution took an average of 17.32s and with 32 units an average of 1.34s, this is a speedup of the factor 13. With more units the average execution time for the *BB* layout no longer decreases and the execution times of the *BN* and *NB* layouts increase. The deviation between minimum and maximum execution time for the copy task is also increasing with more units. Around 32 nodes something in the whole system happens to stop every layout to scale furthermore. To evaluate this further the results are analyzed in more detail. The second implementation (v2) allows to measure the time of copying the data to local, calculating, and copying the data back to the *DASH* array separately. For the test with 128 units and the *BN* layout it took an average of 0.007s to copy the data and halo to the local buffer in comparison to the *NB* layout which took an average of 3.98s. So the *NB* layout spent the most time of the execution on copying the data to a local buffer. This is suspected because of the large stencil communication overhead. Furthermore for the layouts *BB* and *NB*, a decrease in scaling with more units was observed. Also, with *NB* and 128 units the execution time is nearly as bad as just with 2 units. The total execution time for *BN* with 128 units was 1.36s compared to *NB* with 128 units and 19.94s. The decrease of *NB* was expected due to higher stencil communication overhead, but not the decrease in execution time of the other both layouts.

The second test with *ST2*, seen in Figure 5.10 middle plot, shows similar performance on

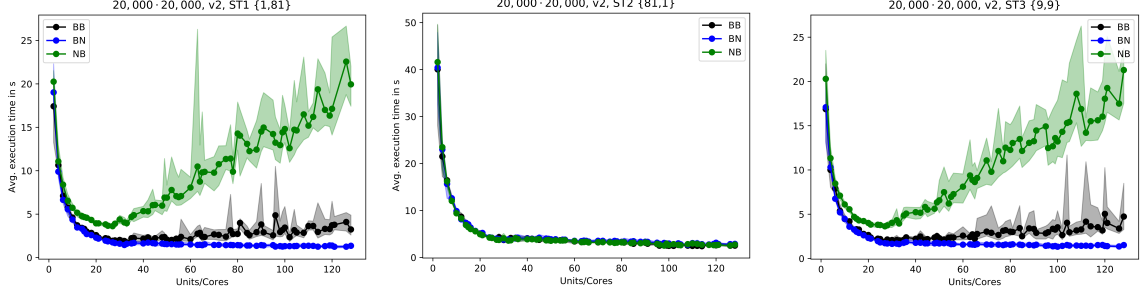


Figure 5.10: Average execution time in seconds of $ST1$, $ST2$ and $ST3$ with 2 nodes with implementation v2 on the *HR-Cluster*.

every layout even though NB was expected to be the fastest. Similar to the first test at around 32 used *units* the system stopped to scale. The total execution time for BN with 128 *units* was 2.89s compared to NB with 128 *units* and 2.75s. On the third test with $ST3$, seen in Figure 5.10 right plot, the BN layout performed best again, even though BB was suspected to be the fastest. Instead BB is slower than BN but still faster than the NB layout. Also the same effect as in both previous tests was observed: At 32 *units* the system stopped to scale. Furthermore for the layouts BB and NB a decrease in scaling with more *units* is observable, again.

Regardless on which stencil and layout the test is running, the layouts start around 32 *units* to not scale anymore. Instead there is a larger jump upwards in execution time and a steady increase with increasing amount of units. A first guess for the rise on every tested layout was the general workload on the system from other users, but in this case the plot would be more volatile and repeated experiments on the system show the same behavior. The parallel workload on the system is an issue while doing performance tests, for a more detailed result a test on a exclusively used resources is necessary. The *SuperMUG-NG* could provide this, as can be seen by the experiment at the end of this Section. One possible reason for the stop of scaling could be a general resource bottleneck in the *HR-Cluster* environment or additionally the communication starts to become the dominant part of the execution time which could be result in a bottleneck in the network. For further investigation, the execution time of the experiments is split up to the three parts: (1) Copying of the data from the *DASH* array to a local buffer. (2) Calculating with the stencil on the local buffer. (3) Copying the data from local buffer back to the *DASH* array.

In Figure 5.11 first plot, the average copy time per *unit* for $ST1$ on the different layouts is depicted. With more *units* the copy time increases in the same way as the total execution time. At the point of 36 units, the time for copying of 3.54s surpasses the calculation time of 1.82s, shown in Figure 5.12 and starts to dominate the execution time for the NB layout. Similar but less strong effect was observed with the BB layout, as shown in Figure 5.11. The NB layout does not produce a large overhead while copying data with a stencil aligned to it, because less network communication is necessary between the nodes. In Figure 5.11 second plot, the average copy time per *unit* for $ST2$ on the different layouts is shown. The copy time for this stencil on the BB layout shows the best results. Depending on the $R_{S/V}$ ratio of the blocks the time is slightly changing. The copy time for NB is fast this time, because the layout does not produce a large overhead while copying data with a stencil aligned to

it. Figure 5.11 third plot, depicts the average copy time per *unit* for *ST3* on the different layouts. The result looks similar to the test with *ST1*. The copy time of the *NB* layout starts to be the dominant factor in the total execution at 36 units.

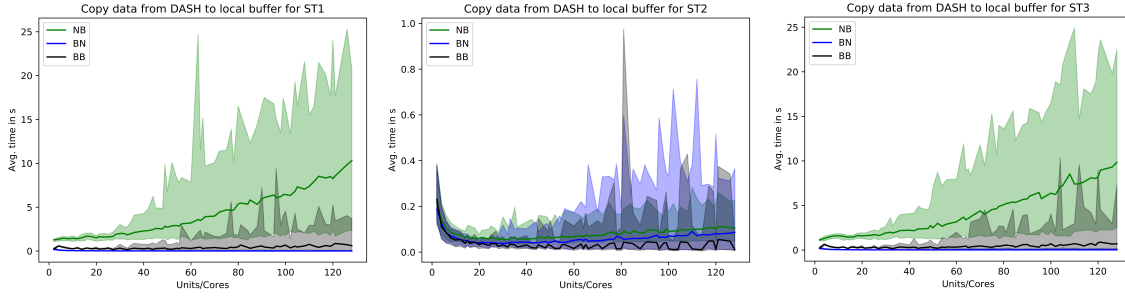


Figure 5.11: (1) Average time for copying a *DASH* array to the local buffer in seconds for the *ST1*, *ST2* and *ST3* stencil. The experiment facilitated implementation (v2) running on 2 nodes of the *HR-Cluster*. The confidence intervals show minimum and maximum copy times.

The Figure 5.12 show the average calculation time per *unit* for the three stencils on the different layouts. Because the application operates on a local buffer, no large difference in calculation time between the experiments was observed. The calculation time for *ST1* with 2 *units* is 18.3s and with 128 *units* 2.05s. This equals a speedup of the factor 17.85. The calculation time for *ST2* with 2 *units* is 17.5s and with 128 *units* 0.98s, resulting in a speedup of 8.93. Finally, *ST3* has a calculation time with 2 *units* of 21.1s and with 128 *units* of 0.90s. The corresponding speedup factor is 23.44. The speedup is at this point far away from the perfect speedup of 64, at this point a optimization of the calculation could bring more speedup, but this is out of the scope for this thesis and is not directly related to the locality behavior. Also the only effect could be the size of the local buffer which have different sizes with different stencil extents, which will again have an impact on the cache miss rates. We can conclude, there is no large deflection in the calculation plots which correlates to the large execution times and so it can be excluded from the search of the issue with decreasing performance.

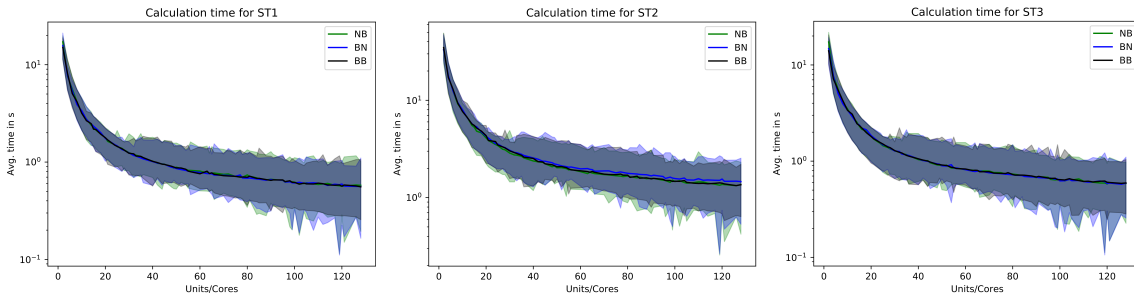


Figure 5.12: (2) Average time for calculating in seconds for *ST1*, *ST2* and *ST3*. The experiment facilitated implementation (v2) running on 2 nodes of the *HR-Cluster*. The confidence intervals show minimum and maximum calculation times.

5 Results & Discussion

In Figure 5.13 the average time for copying the local buffer back to the *DASH* array per *unit* for the three stencils on the different layouts is shown. Writing back to the local part of the *DASH* array does not involve network communication, so each stencil shows similar results. While the copy time of both layouts *BB* and *BN* decrease with more units, the *NB* layout stops to scale at 36 units. This could be coupled to the same issue with copying the data from the *DASH* array to the local memory. This is discussed in detail in Section 5.4 at the end of the chapter.

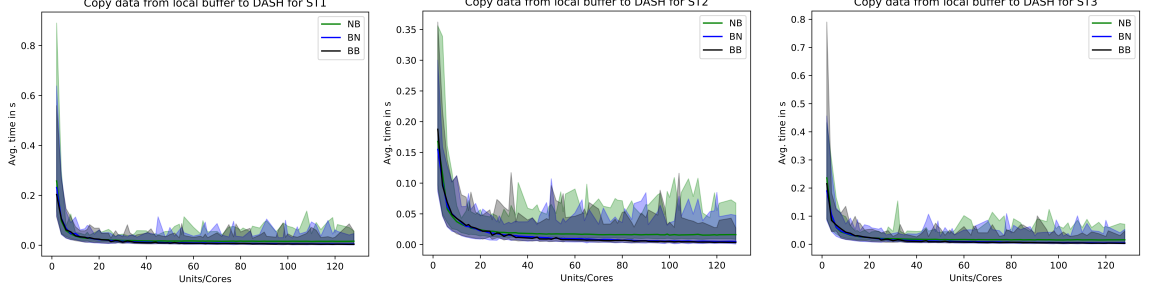


Figure 5.13: (3) Average time for copying the local buffer to a *DASH* array in seconds for *ST1*, *ST2* and *ST3*. The experiment facilitated implementation (v2) running on 2 nodes of the *HR-Cluster*. The confidence intervals show minimum and maximum copy times.

After a more detailed analysis of the complete execution time it is clear that the calculation time and the copy back to dash time is not the reason for stopping the scaling, it is the copy at the beginning which saturates the network with every layout. If this is true, the *SuperMUC-NG* should give more stable result, because of the other network technology and better inter-connect.

From the perspective of total performance the implementation v2 outperforms v1, the result can be seen in Table 5.7 listing the average execution time (avg. exec. time) for 8, 16, 32 and 64 *units* for each stencil with the fastest layout. In the last column the speedup from implementation v1 to v2 is shown. For this excerpt the best speedup was 158.07 and the worst 40.36.

Stencil	Units	Avg. exec. Time v1	Layout v1	Avg. exec. Time v2	Layout v2	Speedup v1:v2
<i>ST1</i>	8	305.37s	BN	2.78s	BN	109.85
<i>ST1</i>	16	161.07s	BN	2.66s	BB	60.55
<i>ST1</i>	32	85.73s	BN	1.25s	BN	68.58
<i>ST1</i>	64	56.13s	BN	0.67s	BN	83.78
<i>ST2</i>	8	335.51s	NB	6.12s	BB	54.82
<i>ST2</i>	16	177.81s	NB	2.93s	NB	60.69
<i>ST2</i>	32	89.42s	NB	1.62s	NB	55.20
<i>ST2</i>	64	57.72s	NB	1.43s	BB	40.36
<i>ST3</i>	8	663.86s	BB	4.19s	NB	158.44
<i>ST3</i>	16	344.92s	BB	1.90s	BB	181.54
<i>ST3</i>	32	176.11s	BB	1.88s	BB	93.68
<i>ST3</i>	64	115.39s	BB	0.73s	BN	158.07

Table 5.7: Example execution times for both implementations on several different tasks with 2 nodes and a 20.000 * 20.000 pixel image. [N = None, B = Blocked]

The experiments were also conducted on the *SuperMUC-NG* to have a more homogeneous system with a faster network interconnect. Another benefit of the *SuperMUC-NG* is that the task schedule assigns node resources exclusively. So no disturbing workload on the same CPU resources from other applications occur while running the experiment tasks. Each test was executed 10 times.

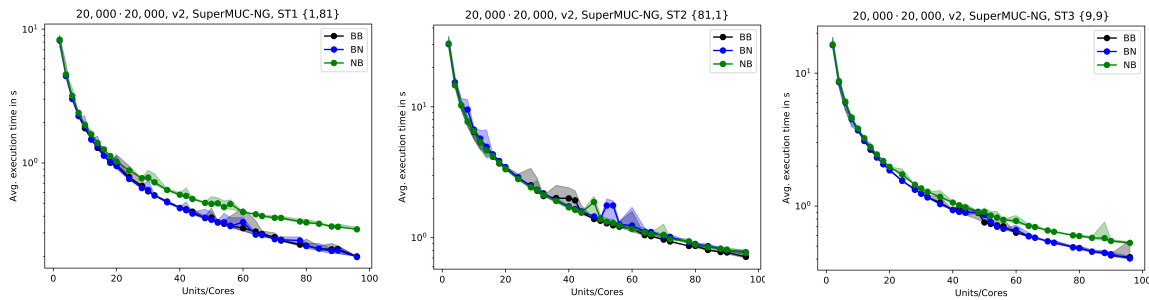


Figure 5.14: Average execution time in seconds of *ST1*, *ST2* and *ST3* with 2 nodes with implementation v2 on the *SuperMUC-NG*. Confidence interval show min and max execution time.

The results of the *SuperMUC-NG* experiments are shown in the Figure 5.14. In comparison

to the experiments on the *HR-Cluster* the results are more stable now. The results of the tests running on 2 *units* differ from the results of the tests running on 96 *units* with regard to speedup for each stencil: *ST1* achieved a speedup of 40.09, *ST2* achieved a speedup of 43.03, and *ST3* achieved a speedup of 40.27. The speedup of every stencil is near to the perfect speedup of 48 and - as seen in Figure 5.14 - the layout does not have a strong impact. Only the *NB* layouts tend to be slower for *ST1* and *ST3*.

A possible explanation for this characteristic is that the workload caused by arrays of the tested size is too small to see scaling problems. To test this assumption, the problem size was increased in each dimension by the factor 10 resulting in a $200.000 \times 200.000 = 38\text{ GB}$ pixel image while reusing the stencils of the previous experiments. In addition to that the number of nodes was increased by the factor 8 to 16 nodes. To stay below the timeout limit of the test partition on the *SuperMUC-NG*, the iterations were limited to 3 iterations for tests with less than 10 *units* per node. Figure 5.15 shows the result for *ST3*. As for the previous experiments, the results show a good scaling and no huge impact of the different layouts. Only the *NB* layout is a little bit slower again.

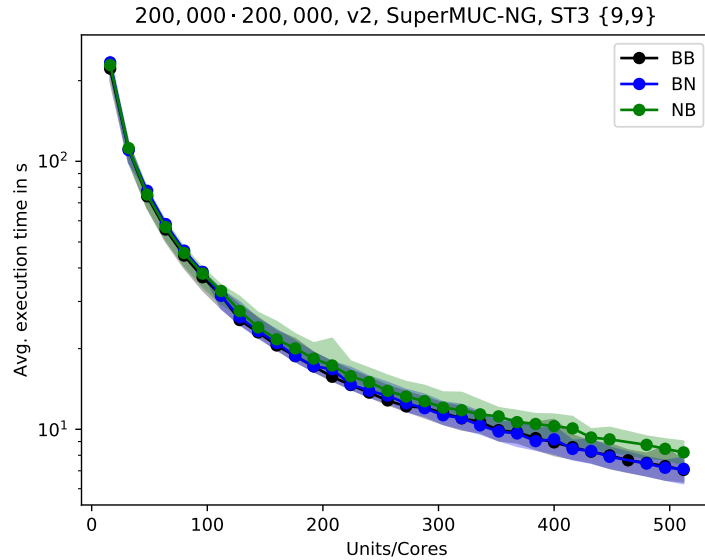


Figure 5.15: Average execution time in seconds of *ST3* with 16 nodes with implementation v2 on the *SuperMUC-NG*. Confidence interval show min and max copy time.

Another assumption would be the stencils are not large enough, so we increase the stencil *ST3* to 500 in *x* and 500 in *y*. But if the stencils is increased further the difference between *NB* and both other starts to increase, see Figure 5.16. With 512 *units* the *NB* layout took in average 0.76s with a median absolute deviation of 0.071s, *BN* took in average 0.12s with a median absolute deviation of 0.0046s and *BB* took in average 0.073s with a median absolute deviation of 0.0041s to copy the data from the *DASH* array to the local buffer. The *BB* layout is 10 times faster than the *NB* layout. The median absolute deviation is about the factor 18 higher with *BN* in comparison to *BB* and is nearly equal to the average copy time of *BB*. For other experiments with larger *ST1* and *ST2* the results look similar. Below 288 units, the *BB* layout is slower than the *BN* layout.

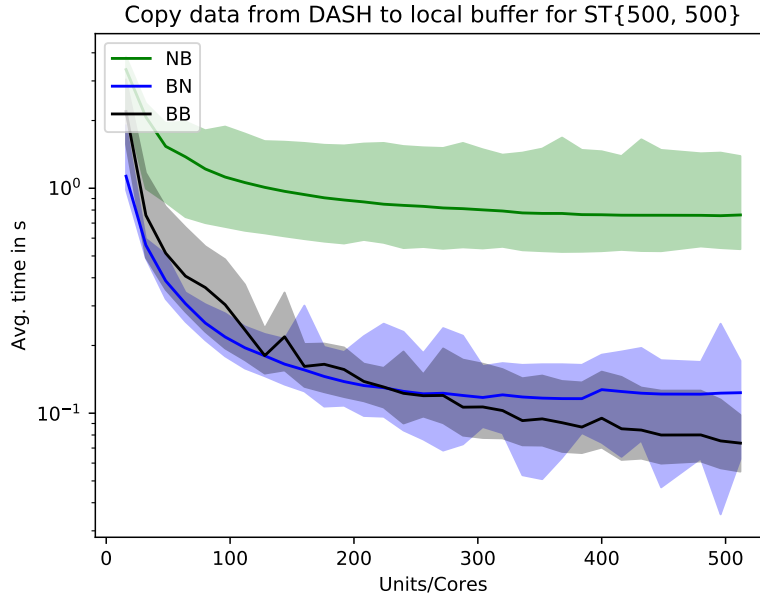


Figure 5.16: (1) Average copy *DASH* to local buffer time in seconds of $ST\{500, 500\}$, 16 nodes and implementation v2 on the *SuperMUC-NG*. Confidence interval show min and max copy time.

The determination of the weights from experiment with implementation v1 makes no sense, because the overall execution speed was too slow to perform real experiments to validate them in context. With the v2 implementation, the execution times of large arrays are faster, but the *NB* layout is always faster than the other two. From this perspective it would be better to store all data line wise and use every time the *NB* layout, more in the following Discussion.

To demonstrate that the application is also working with a very large array and a larger stencil, as proposed in the introduction, implementation v2 was used to process an image of $275.000 * 275.000$ pixel on 96 *units* with a *BB* layout. The 70 GB image was processed on 3 nodes of the *HR-Cluster* (*HR-SLX005*, *HR-SLX007* and *HR-SLX012*) as well as on the *SuperMUC-NG*. A stencil size of $21 * 21$ was used. For the test with the *SuperMUC-NG* the layout *BN* was used. The test was limited to 70 GB because of the memory limitations of 90 GB for executions on the test partition of the *SuperMUC-NG*. A margin of 20 GB was left to account for the memory overhead of the *MPI* framework and the local buffers on the single nodes. On the *HR-Cluster* the test took 344s and 96.933.790.259 cell updates per second were achieved. The copy process took 0.55s for the fastest and 1.46s for the slowest unit. The calculation took 190s for the fastest and 339s for the slowest unit. The copy of the results back to the *DASH* array was achieved in 0.38s for the fastest and 1.55s for the slowest unit. On the *SuperMUC-NG* the test took 305 seconds and 110.386.788.754 cell updates per second were achieved. The copy process took 0.46s for the fastest and 0.53s for the slowest unit. The calculation took 249s for the fastest and 301s for the slowest unit. The copy of the results back to the *DASH* array was achieved in 0.60s for the fastest and in 0.65s for the slowest unit.

From the comparison of the tests on the *HR-Cluster* and the *SuperMUC-NG* it can be seen that the *SuperMUC-NG* is approximately twice as fast while copying data, and the calculation time is 33s faster. The *HR-Cluster* is slower and shows an efficiency of $E(I) = 88\%$ compared to the *SuperMUC-NG*. Also the deviation between the slowest and fastest *unit* is smaller on the *SuperMUC-NG*. The reason for this is the homogeneous system in comparison to the node variety in the *HR-Cluster*. But with this test it was proofed that a execution on large images its possible on both systems. Due the issues with the performance and missing real workload for the calculation part an extraction of weights was not possible, more in the Discussion.

5.3.4 Weak/Strong Scaling

For testing weak and strong scaling implementation v1 will be first discussed based on the results on a single node followed by tests in a multi node environment. This test is for observing if the application is scalable in general and how the algorithms react to strong or weak scaling. For workload simulation the same stencils as before are used as listed in Table 5.3. This test are not directly related to *brick locality* more to *data locality*, but from strong deflections in the graph problem with the general implementation can be derived. The three stencils are considered as a single job. For each stencil the *DASH* memory layout with the best fitting configuration from the experiment in Section 5.3.3 is used. All data for the simulation is generated on the fly. In weak scaling the problem size is fixed to 1,000,000 pixel for each unit. For strong scaling the same data size as in the experiment of Section 5.3.3 is used. For multi node tests, the default *unit* pinning of the *DASH* framework will be applied, which means that the *units* are distributed to the nodes in a round robin manner. For the test on the *HR-Cluster* all available nodes are used while 16 nodes are used for the test on the *SuperMUC-NG*. The tests for the single node computing are limited to 64 units, because this is the maximum number of cores the infrastructure covers on the available single node system (*HR-SLX007*).

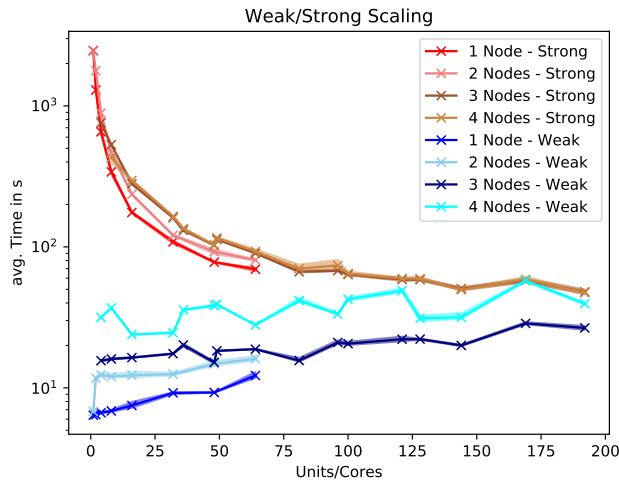


Figure 5.17: *Weak/Strong scaling* shown on a single node and in multi node environments with implementation v1.

Figure 5.17 depicts the average execution time with regard to the increasing number of cores. The red lines show that the strong scaling is decreasing the execution time with more cores. In contrast, the blue lines show that the weak scaling is slightly increasing the execution time, because of the additional communication borders which come with more nodes. At the upper end of *units* the weak and strong scaling is starting to become more and more equal. The reason is that at this point the problem size per *unit* for the strong scaling is nearly equal to $1000 * 1000$ which is the problem size of the weak scaling test.

In Figure 5.18 the weak scaling results of four nodes divided for each stencil with implementation v1 are shown. The overall average execution time of Figure 5.17 is strongly affected by the bad execution time of the *BB* layout seen in Figure 5.18. It can be seen that

the execution times correlated to the effect of *surface-to-volume* ratio and the estimation of the cost functions, see Figures 4.4.

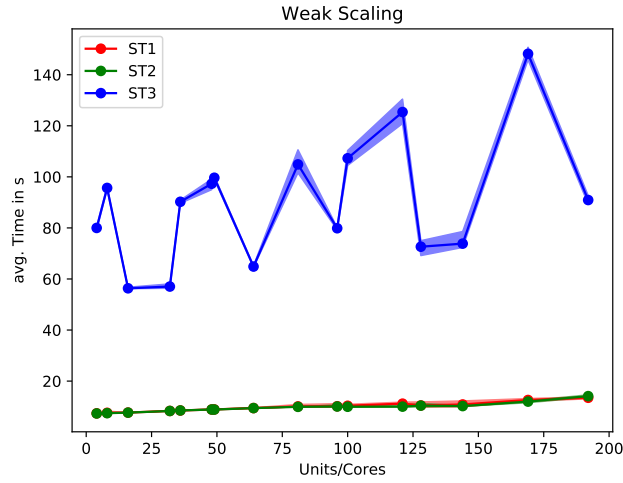


Figure 5.18: Weak scaling shown on four nodes divided for the different stencils with implementation v1.

For implementation v2 *NB* seems to be the fastest layout for every stencil, so every stencil is executed with this layout. This test was only executed on one to three nodes, because of troubles while executing implementation v2 on *HR-Test001*. The results look similar to the v1 results, but because of the overall faster execution time the number of *units* to test was increased for strong scaling. Again at the upper end of *units* the weak and strong scaling test become more and more equal due to the same reason as for the v1 experiments described earlier. Unfortunately, the test was run while other users are also beginning to perform tasks on the system, so the results cannot be considered reliable, see Figure 5.19. But the trend looks similar to the trend in implementation v1.

The test environment is switch to the *SuperMUC-NG* for a more detailed and more reliable test. This systems provides a faster interconnect and a homogeneous system with more resources. Again the test is only applied for the fast solution with implementation v2, because the first test with implementation v1 shows a similar performance and a reevaluation was deemed to be too time consuming. For the strong scaling test, the result from the layout performance test before with 16 nodes and a $200.000 * 200.000$ large array was used as depicted in Figure 5.15. The weak scaling is limited to $1000 * 1000$ pixel per core. The result for weak and strong scaling is shown in Figure 5.20. The weak/strong scaling is split up for each stencil. Like for the previous experiments the weak and strong scaling results cross at the upper end of *units* because of the equal problem size per unit.

From this result we can conclude there is no strange behavior in the implementation, which would result in a deflection in the execution time for certain units and stencils combinations. Also a quite good speedup of 29.86 was achieved by the *BB* with *ST2* layout. This is an increase in speed from a total execution time 107.01s with 16 units to 3.58s with 512 units, this is near to a perfect speedup of 32.

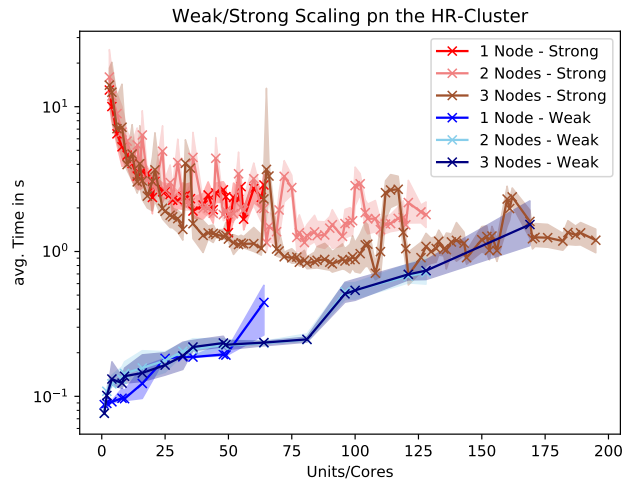


Figure 5.19: Weak/Strong scaling shown on a single node and multi node environments with implementation v2 on the *HR-Cluster*.

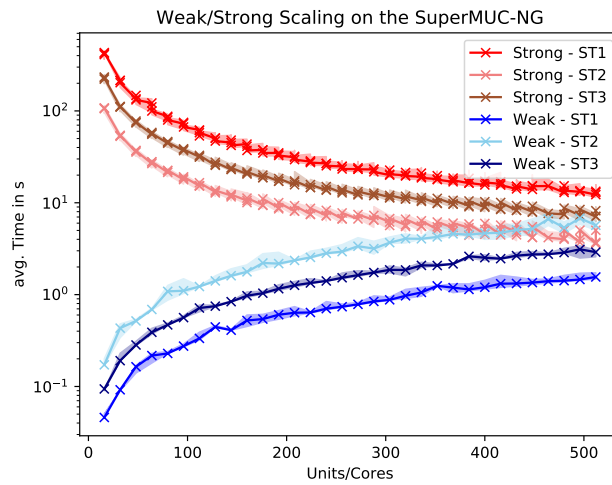


Figure 5.20: Weak/Strong scaling shown on the *SuperMUC-NG* with implementation v2 on 16 nodes divided for the different stencils.

5.4 Discussion

Using the parallel disk speed of all participating nodes to initialize a large *RS* image and write the results back is a promising approach, as introduced in Section 3.7. The approach allows to increase the read/write performance with each new node that is added to the system making it superior to a central *NFS* system. Additionally every new node also increases the amount of cores and thus processing speed. An important thing to keep in mind when considering such a system is that the network becomes the bottleneck for the speedup in the initialization phase. The main disadvantage of the system is the costly task of distributing the image in bricks to the nodes and collecting the results. Even though this task is very time consuming and the data handling can be complex in a distributed system, it could pay off fast because of the additional computing resources available for the processing task. It is assumed that in the best case the *brick locality* matches up with the best layout for the first stencil operations on the data. At this point of the investigation it was assumed that the general processing steps for raw data (transformations) are similar and only for high-level products (fusions) the used stencils become more volatile. So this could be true. The *brick locality* test of this thesis shows promising results with a speedup by the factor of 6 in the best case when reading from a local disk with 100% brick locality, and by the factor of 2 in the worst case with 0% locality. From the perspective of only *brick* loading times without comparison to the *NFS* the 0% brick locality to 100% brick locality its about 25% faster. The results of the experiments showed that it is possible to speedup the complete initialization process by partitioning the data to all nodes and read from local disk during processing.

If the storage layout don't matches the *PGAS* layout of the initializing step and the following stencil operation, there will be network communication due to the *bricks* which have to be transferde over the network while initializing and the stencil communication overhead. So a a strong network backbone is still necessary, as the experiments with v2 shows that the *HR-Cluster* stop to scale in comparison to the *SuperMUC-NG* with a much faster inter-connect. For the *HR-Cluster* environment, the network saturation was quickly reached because it facilitated only a flat network hierarchy with *10Gb Ethernet* compared to the *100 Gb Omnipath Fat Tree* on the *SuperMUC-NG*.

If both the network backbone and the parallel file system in the background are powerful enough, it would be the preferred solution to load the required data from the parallel file system because it is less complex to handle the data and unoptimized applications will benefit as well. On the other hand, the distributed loading of data on the given infrastructure works only with up to 64 *units* without running in the network bottleneck. This thesis has shown that this is especially true for the loading of a brick segmented image is loaded to a *NB* layout which generates much more *MPI* calls compared to the other layouts.

For validated the implementation, finding weights for the cost function and proof the assumption that the execution time can be accelerated if the stencil matches the memory *DASH* pattern, memory layout performance tests with the three different stencils were executed. The stencils were aligned in size to stencils which are used in applications for real-world *RS* image processing. With the first implementation v1, every stencil behaves on the layouts as predicted: *BB* worked the best for a stencil with equal extents in each dimension, *NB* was the fastest with the largest extent on the first dimension, and *BN* outperformed the other layouts for stencils with the largest extent on the second dimension. These results showed that it is worth optimizing the *DASH* memory layout to the used stencil to improve the pro-

cessing performance. In addition to the expected influence of the combination of the layout and the stencil size on the performance, the evaluation experiments also exposed a non-optimal behavior of implementation v1 resulting in large performance differences between the total execution time of the different stencils.

A first explanation for the slow of the execution times between *ST1*, *ST2*, and *ST3* was the cache miss rate. Evaluation experiments conducted in this thesis to verify this explanation based on different stencils and *unit* combinations did not showed a strong impact on the overall performance. This result was expected because *DASH* already takes care of cache access in its framework. Finally, a more satisfying explanation of the behavior was found: The main performance impact is caused by loading pixels which are not locally available using single *MPI* calls and going through the whole software stack build up by *DASH*.

Finally, a more satisfactory explanation for the behaviour was found: The main impact on performance is caused by loading pixels by working on the global *DASH* array. First, for every single pixel which is not available locally, the access results in a background *MPI* call while the processor waits for the result of the call and is not able to calculate more data. On the other hand for each pixel the software stack (*DASH*, *DART*, *MPI*...) is run through once, which adds a delay on the software side. Because of this, the *unit* spends a lot of time waiting for remote pixels which slows down the complete calculation. If no network communication is necessary the execution time is much faster as demonstrated by *ST1* with *BN* and *ST2* with *NB* depicted in Figure 5.7 but still slow because of the software side latency. At the first sight, the total processing time for both stencils using 128 cores looks quite fast with 45s, which is a speedup of about the factor 100 compared to the solutions which generate the most network communication. But only 1.2 GB of data were processed with 128 cores, which is not a lot of data in this time frame. It turns out that work on the global *PGAS DASH* array is terribly slow. This was the reason to investigate an alternative implementation v2 which copies the local data sets with a *halo* region to a local array and is able to calculate using this array without any *MPI* calls.

The results of the experiments with implementation v2 show an improved overall performance compared to implementation v1. The fastest execution time was achieved with layout *BN*, regardless which stencil was chosen. At the point of 32 used *units* the *HR-Cluster* stops to scale further. As a reason he bottleneck in the network infrastructure preventing the application from scaling up was estimated. ut this can not be the only reason for the bad performance of *NB*, because similar network issues would be expected when using a *BN* layout with a not aligned stencil. Furthermore, the slightly worse execution time on the *SuperMUC-NG* would not be explained by this. So what is the difference in the execution between the *BN* and *NB* layout? It was observed that the issue lies inside the function *copy_dash_to_local* of implementation v2. In the main part of this function, the data is copied from the *DASH* array to the local buffer in a line wise manner. For the *BN* layout this means that the function will copy full lines of the image, so each *unit* will generate $\frac{ext_x}{u}$ copy calls each with the size of ext_x pixels. For the *NB* layout, each *unit* will generate the same amount of copy calls, in particular the count of pixels in the first dimension, regardless of how many *units* we are added to the problem solution. The calls will shrink in size by $\frac{ext_x}{u} + sext_y$. The $sext_y$ stays the same, with increasing number of *units* and this result in more data to transfer over the network, because the *halo* become large in relation to the shrinking block size in y . The amount of copy calls remains the same. The correlation is shown in Figure 5.21. Again each *dash::copy* call will also add the software-side latency to the total execution time.

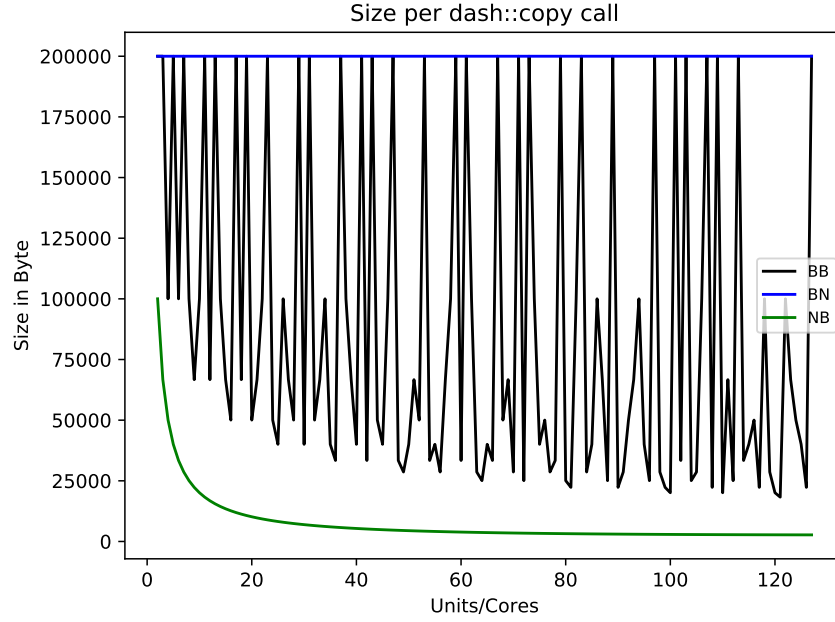


Figure 5.21: Comparison of the amount of byte which are copied per `dash::copy` call between the different layouts [B = BLOCKED, N = NONE]

For the *HR-Cluster* we suspect the problem with the performance of small copy calls is due to the network protocol *transport control protocol* (*TCP*) and also through the latency stack which results from *DASH* and *MPI* because each call has to go through the whole software stack once through. On the basis of this assumption the estimated communication on the *TCP* level in the network for the *HR-Cluster* is calculated and shown in Figure 5.22. As parameters the *maximum transmission unit* (*MTU*) of 1500 is assumed and for simplification the overhead from the *MPI*- or *IP*-Headers is ignored. The Figure 5.22 shows how network communication with an increasing number of *units* under different patterns develops with the *ST3*. In the left plots, it can be seen that the total amount of data to transfer and resulting *TCP*-Segments strongly increasing with more *units*, in the case of the *NB* pattern. On the *BN* pattern everything stays constant. The *BB* patterns shows a slightly increase in data to transfer. On the right side at the top, the utilization of the maximal *MTU* size is shown. The *BN* and *BB* pattern are resulting in large calls and so they can utilize the network better in comparison to the *NB* which generates much smaller calls than the *MTU* size. This estimation is not true for the *SuperMUC-NG*, the network is not based on *TCP* communication. But the statement with the increasing data to transfer, and amount of calls is also valid for the *SuperMUC-NG*.

In the end it is better to make fewer calls copying larger data chunks instead of more calls copying only small parts of the data. The reason is a better network utilization with larger calls, because smaller calls generate more overhead by not fully filling the data packets as well as adding the communication latency of all calls through the whole software and hardware network stack to the execution time. The *SuperMUC-NG* does not use *TCP* so only the accumulated software latency stack seems to be the reason why the *NB* layout is slightly worse than the other layouts on, despite the faster and strong interconnect. But it is to

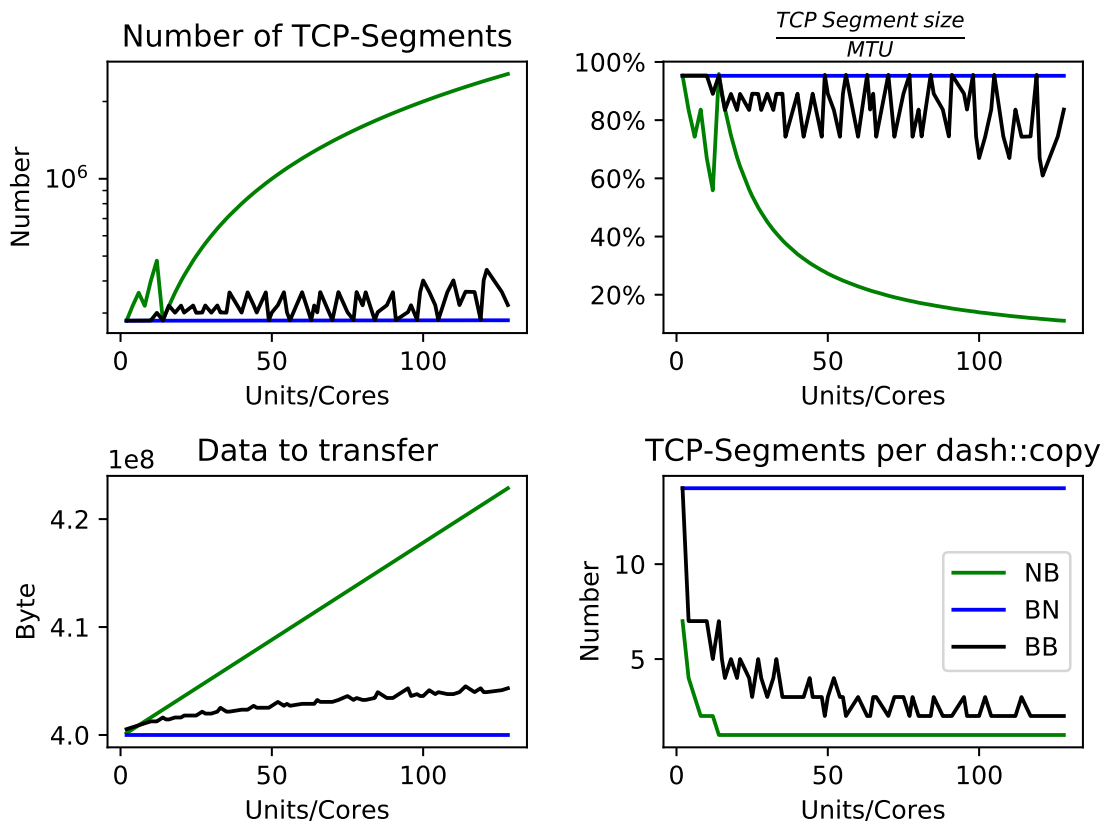


Figure 5.22: Calculated behavior of the *TCP* traffic on the network with a *MTU* of 1500. Top left: shows how many *TCP-Segments* are generated in total. Top right: show the utilization of the *MTU* size. Bottom left: the amount of data to communicate. Bottom right: shows the number of generated *TCP-Segments* per *dash::copy* call

mentioned, that the other technology of the network in the *SuperMUC-NG* can also have other influences. It can be concluded that with the *BB* and *BN* layout it is possible to reduce the amount of calls with an increasing number of *units* to be able to transfer more data per call which results in a better performance than with the *NB* layout.

This results in the open issue remains the chosen interaction between the *DASH* framework and the implementation of the *copy_dash_to_local* function. The way of copying the *halo* for *NB* layout have to be improved to be able to weight the execution times of the stencils in relation to the layouts correctly. The implementation can be improved by adding a *halo* synchronization which copies the data in a bulk, so only one large copy call for the halo is generated, which is split up to the 8 neighbors of the unit. This would allow the *DASH* framework to handle the *MPI* calls in an optimized way in the background. In a next step the implementation can be extended to allow the calculations to work on the local pixels while the *halo* region can be synchronized in parallel in an asynchronous task, which will hide the communication latency. This would hide a part of the latency of accessing the remote pixels by calculating in parallel on the local pixels. The *DASH* framework will provide this feature in a future release.

A theoretically solution for not using the the *NB* layout to avoid the discussed limitations, is to use the *BN* layout instead on a transposed image with a transposed stencil. This approach results in improved execution times for the *ST2* stencil. Of course transposing the image generates additional overhead, but this is neglected for this theoretical view on the problem. The results of applying this solution to the evaluated performance values of the *copy_dash_to_local* function for the *ST2* stencil on the *HR-Cluster* are depicted in Figure 5.23. Through transposing both the image and the stencil, *ST2* is aligned to the layout and the copy process performs much better now allowing to have similar execution times as *ST1* on a *BN* layout. But still the overall network saturation and latency is an issue on the *HR-Cluster* compared to the *SuperMUC-NG*. Especially for a *Beowolf* like cluster, a bulk copy should bring a better network utilization compared to line wise copying.

In the end the line wise copy remains a main issue of the current implementation both on the *BB* as well as the *BN* layout. For the *BN* layout the implementation for a bulk copy is trivial because the data to copy is stored consecutively in the memory and so a copy of all lines in one large call is possible with *dash::copy* allowing *DASH* to optimize the copying in the background. First evaluation experiments showed no large performance gain which is was attributed to the fact that the existing calls were already big enough to utilize the network. Implementing the solution for the other layouts is out of scope of this thesis because of the more complex implementation that needs to consider the none consecutively memory access in the *DASH* array. In the next *DASH* release a feature for *halo* synchronization will be supported which generating a separate memory space for the *halo* which can be synchronized in the background while calculating on the local data.

The results show that with each added *unit* the application generates more communication and so the total execution slowly grows. Because of this, a perfect weak scaling is not achieved. This is expected as demonstrated by the experiments on the *SuperMUC-NG* in Figure 5.20. But nevertheless perfect weak scaling is not achievable at all, due too *Amdahl's Law*. To come near the perfect scaling, this could only be possible if no neighbor communication happens. In this case the neighbor communication is increasing with the amount of *units* and this results in an increased total execution time from 1.15s with 16 *units* to 3.38s with 512 units on the *SuperMUC-NG*.

For the strong scaling the result looks different. With an increasing number of *units*

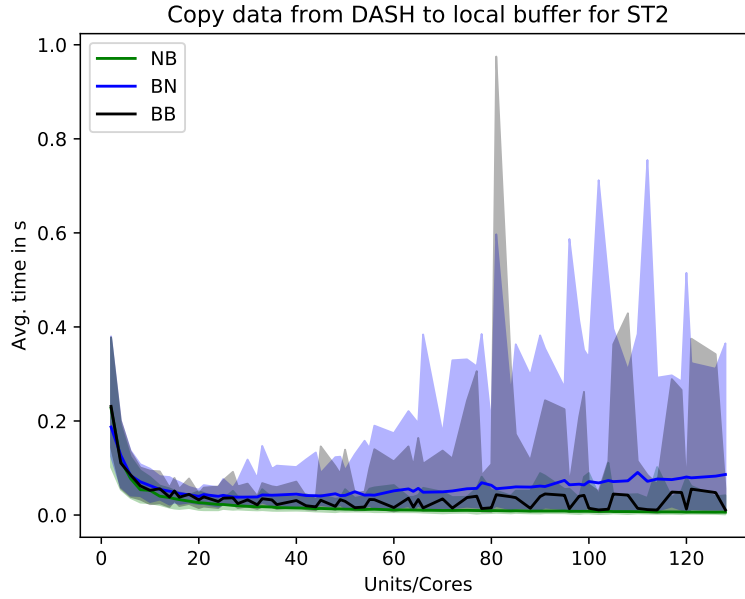


Figure 5.23: Theoretically layout performance for ST2 on 2 nodes in the *HR-Cluster* with transposed data.

the workload per *unit* is decreasing and dominates the increasing stencil communication overhead. Within the strong scaling experiment a speedup of 24.16s from 16 *units* to 512 *units* was achieved on the *SuperMUC-NG*, where the perfect speedup would be 32. The speedup could be further increased with a more optimized calculation step, because the largest part of the execution time is spent in the calculation step. But for this purpose first real *RS* algorithms have to be implemented, otherwise this is a pointless effort.

A focus of this thesis lied on the proposed optimization model, which makes a minimum optimization for the network communication which occurs in the whole multi-channel *RS* image processing chain. The network communication includes the initial loading of the individual *bricks* into the *PGAS* shared memory as well as the resulting communication during the calculation, caused by the potentially large stencils. From this starting point, an optimization model was created that determines optimization costs by the amount of transferred pixels. Each pixel which needs to be transmitted over network between nodes while calculating or switching the layout increases the total cost while all local pixels reduce the total cost. Inside the optimization model the layout switching cost and communication cost are treated equally. In the experiment executed in Section 5.3.2 it was found that the resulting costs, see Figure 4.3, of the implementation v1 matches in the trends with the execution times. We can only compare the trends because of it is not possible for the cost function to reflect the whole behavior of the hardware and software stack, its just a simplification. The only concern of the optimization function is to reduce communication.

Because this implementation does not perform good on larger data sets, the experiments were conducted using only a small data set. For the used stencils *ST1*, *ST2* and *ST3*, the optimization model was dominated by the choice of the layout in the first processing step. This layout was propagated through the complete processing chain regardless of which layout

will result in the best performance. If the initialization locality was ignored or the stencils communication overhead were enlarged, the layout changes fitted the best execution times for the processing step.

It can be concluded from the first experiments in Section 5.3.2 that the optimization model described in Section 4 has a few weaknesses: The locality of the initial distribution has a too strong impact on the first layout in comparison to the stencil communication overhead. (2) If the stencil is not large enough, the model tend to do not change the layout inside the processing chain. One solution for both (1) and (2) is to weight the cost for initializing data/switching the layout while processing and the communication costs of the *halo* regions differently. In order to do so, we tried to estimate weights in the Section 5.3.3 but its was not possible due to performance issue in the *HR-Cluster* while scaling up further then 36 units and the overall better performance of *NB* on the *SuperMUC-NG*. If the results of the *SuperMUC-NG* experiments applied into weights it is suspected that the optimization function will show to use the *NB* pattern in the most time and it would also the best to store the data directly line wise to match the *NB* layout. At this point we could conclude, we find out that storing the data *line wise* is the best option. But the generated weights are strongly effected by the kind of implementation and the evaluated problems with v2. So to make this conclusions, the issue have to be removed from the implementation first. Also, it would the best to generate respective weightings with real *RS* image processing to have real context between calculating time on the data and the time for loading the data from the disk with initialization to a *PGAS* memory to be able to draw this conclusion.

But we can conclude the optimization function really reduce the network traffic in the context of pixel which have to transferred over the network, nevertheless it results in bad execution times. But for a real proof a measurement of the network traffic in an enclosed environment is necessary, which could not be achieved while the evaluation of this thesis.

Anyway the implementation v2 even showed the capability to process large data sets by processing an evaluation data set of 70 *GB* in only 305 seconds allowing to use this version in future real-world *RS* image processing applications. But again, the real execution time will differ if we apply real *RS* image processing on the array which will result in another algorithm complexity then just a smoothing and so we suspect higher execution times.

Another result of the evaluation experiments is that the total execution time of the processing task depends on the slowest node. This is because while the slowest node is still working on the task, all other nodes wait at the synchronization point. This is a general problem of this kind of work distribution on a heterogeneous platform like the *HR-Cluster*. On the *SuperMUC-NG* the results were more stable, because of the homogeneity of the system.

6 Outlook - Interaction with Python

The scientists in the DLR *HR* Institute use the *Python* programming language for scientific analysis of data. To allow them to interact with the *DASH* framework, or another distributed computing back-end, the development of a *Python* library is considered. There is already a *DASH* interface library developed by Josef Schaeffer called *pydash*. These *Python* bindings expose the *DASH* data types to *Python* scripts and can therefore serve as a basis for a new interface.

The envisioned light weighted library should implement the following features:

- small programmatic overhead
- interaction with *Python* libraries like *Pandas* and *Numpy*
- abstraction of distributed file handling
- accelerate processing without knowledge of the infrastructure
- set of processing algorithms for radar data

The small programmatic overhead allows the user to process files with only a few parameters, e.g. either the input image name, the task with its stencils and the operation or a *Numpy* array or *Pandas* data-frame with the respective operation. The algorithms are directly implemented in *C++* and will be accessed via a *Python* API. Because most of the RS algorithms are currently implemented using old programming languages like *Interactive Data Language* (IDL) a *C++* re-implementation is inevitable and is expected to be costly. Nevertheless, it is important to have a basic set of algorithms implemented to cover the most RS processing procedures to create acceptance for the newly developed library and facilitate productive use.

The library should support the user to gain as much acceleration from the computation hardware infrastructure as possible while not bothering with the infrastructure itself. On the basis of the input parameters passed to the library, a *DASH* array with the initial data will be created followed by the execution of the operation with the stencil fulfilling the following requirements:

- choose the amount of units to generate a good performance on the given data set
- choose the nodes so that the block locality per node is the highest
- free resources on the node for processing

The library should split up the interaction into three steps: (1) Inputting raw data, segmenting it into multiple bricks, and distributing it to the cluster. (2) Processing the bricks distributed on the cluster inside the desired framework (e.g. *DASH*). (3) Outputting bricks into a product file. Inside the library each file is identified by a unique id. It could be

a benefit to store the raw data in ***Hierarchical Data Format***, as HDF5 [Gro19] files, to make parallel *I/O* available. The raw data as well as the resulting products need to be stored on a persistent system with a reliable backup for data failure protection. This should not be done with the data that is distributed on the cluster.

This abstraction should allow the scientist to easily interact with the *DASH* framework and process tasks in a fast and reliable way. Furthermore, the abstraction allows to change the back-end while keeping the processing scripts of the scientists. This interface with *DASH* as an back-end is envisioned to replace the currently used single node processing frameworks in the future.

7 Conclusion

Current developments, as stated in the introduction, suggest that remote sensing data processing is increasing and becoming an important topic in the world of *Big Data*. However, this results in a variety of implications in efficient management, loading distributed data *I/O*, irregular data access and complex dependencies between several data sets. Furthermore, handling of this fore points together in efficiently way on a operating processing system is a huge challenge. The focus is on techniques to support multi-stage processing chains for extremely large, multi-channel remote sensing data. Particular attention is given to supporting computations that involves potentially large neighborhoods, or stencils. For this purpose, the storing locality of the data on disk in the context of the initial *PGAS* memory layout and of the followed layouts in the further processing chain was investigated. In contrast to traditional cluster computing approaches, which use a centralized network storage such as *NFS*, the thesis investigates forms of distributed storage. The motivation is to improve scalability of the overall system by removing the centralized storage as a potential bottleneck.

The results of the *brick locality* experiments, Section 5.3.1, show that it is possible to improve the initialization speed of large image data by exploiting the local disk speed of multiple nodes in comparison to the initialization of the data images from a central *NFS* server with a *RAID* system in the background. A speedup of 25% was achieved while using 100% *brick locality* against 0% *brick locality*. The 0% *brick locality* is also four times faster than reading from *NFS*. The reason for this, is the gained speed of *brick reading* from several disks outperform the *RAID* system in the background of the *NFS* server. Network bandwidth and saturation is also to concern, but was not the main reason for the better performance in this particular experiment.

Based on the first performance measurements on the layouts, an optimization model was developed with the goal of reducing network traffic. Network traffic occurs in the context of the initialization of a distributed data structure from disk to a *PGAS* memory layout and when performing computations with potentially large stencils on the shared memory layout. The network traffic was calculated based on pixels that must be transmitted over the network when initializing or switching between *PGAS* memory layouts. On basis of this model a *Python* software was developed to calculate the memory layouts with the least network communication cost for a given multi-channel remote sensing data processing workflow and the given data distribution on the nodes. The results of this optimization was evaluated in an experiment, see Section 5.3.2. The evaluation shows that this optimization approach correctly identifies the processing strategy corresponding to the minimum inter-node network traffic. For very small stencils, however, the optimization did not reproduce the real-world optimum because the underlying model is not able to reflect the behavior of the certain types of overhead that occur in the particular implementation of the distributed processing framework. In real world implementations, it might be desirable to extend the optimization model to reflect these implementation-dependent sources of overhead.

In the layout experiments with different stencil extents in Section 5.3.3 the performance behavior of the stencils is analyzed. For these experiments an application, called *Remote*

Sensing Image Distributor and Processor (RSIDP) was implemented in order to process huge arrays on a distributed system. *RSIDP* is based on the *DASH* library, a *C++* Template Library for Distributed Data Structures with Support for Hierarchical Locality for High Performance Computing and Data-Driven Science [FFK16]. During the execution of the experiments several performance issues were solved and the total performance of the application could be improved. Implementation v2 of the software *RSIDP* outperformed implementation v1 by a speedup of 40 to 150 depending on the used stencils and *PGAS* memory layout. This was achieved by using a more efficient way in interaction with the *DASH* framework to handle the inter-node communication at *block* borders in the *PGAS* memory. However, in the end it was not possible to extract weights from the layout performance experiments, see Section 5.3.3, to handle the stencil communication overhead of the different layouts or weigh the stencil overhead against the *brick locality* due to the behavior of implementation v2 (I.e. it performance the best one certain memory layout) and due to missing real remote sensing processing workload.

The general usage of the *DASH* library works very well for parallel processing and generally provides an interface that simplifies the implementation of the distributed algorithms considered relevant to remote sensing applications. This is primarily done by an abstraction layer for the data access simplifying the respective implementation compared to a plain *MPI* solution. At no point during the experiments a problem with the size of the arrays was encountered. Relying on a framework like *DASH* to handle the abstraction of extensive infrastructure in order to process large arrays is therefore a valid approach for future data-intensive remote sensing processing tasks. Nevertheless, the developer still needs detailed insight knowledge on both his own application as well as the utilized framework itself. The issues revealed in the experiments in Section 5.3.3 highlight some of the pitfalls which might otherwise occur.

The existing or envisioned hardware infrastructure thus plays an important role in selecting the appropriate distributed computing framework. The *DASH* framework itself performs best on a homogeneous node and network infrastructure, e.g. a real *HPC* infrastructure like implemented by the *SuperMUC-NG*. On the more *Beowulf*-like *HR-Cluster*, the determining factor of the execution time was the node with the slowest CPU or disk speed because the other nodes had to wait for its operations to finish before continuing the processing after a synchronization point. Data locality will, however, be a consideration in any environment.

In the future the DLR’s Microwaves and Radar Institute will invest more effort in bringing remote sensing processing into the world of distributed computing. Another approach with a *Big Data* framework was already implemented with *SPARK* [Apa20], which also allows distributed calculation of remote sensing data. In the future a comparison between the solution with the library *DASH* and *SPARK* is envisioned. To facilitate this comparison, the next step is to translate part of the *SAR* radar signal processing algorithms from *IDL* or *Python* into *C++*. Beyond the actual comparison, another goal is to speed up the initialization process of *SPARK* with the developed optimization approach as well.

The processing of large data sets remains a challenge because existing solutions are often task-specific or require expert-knowledge of the system and infrastructure. With the given optimization model it is possible to reduce the communication overhead throughout an entire processing chain. However, a further investigation for analyzing the difference between the stencil operations on the given layouts is necessary. The approach to abstract *MPI* with the *DASH* framework shows already promising results, see Section 5.3, towards a more easy to implementation of flexible, generic and reliable solutions. Also loading the data

distributed from several nodes instead of a monolithic network file system approach and the consideration of the *brick locality* while initializing a *PGAS* memory layout show promising results.

This thesis presented some approaches towards such solutions that can be used for real-world remote sensing imaging applications and contribute towards solving the challenges of the modern *Big Data* world for future scientific data exploitation.

Symbols

p	Pixel of an image
lp	Local pixel of an image
rp	Remote pixel of an image
cp	Communication pixel
ext_x	Image extents in x
ext_y	Image extents in y
I	Image consists $ext_x * ext_y$ pixel
$bext_x$	Block extents in x
$bext_y$	Block extents in y
B	Block consists $bext_x * bext_y$ pixel
$brext_x$	Brick extents in x
$brext_y$	Brick extents in y
Br	Brick consists $brext_x * brext_y$ pixel
d	Data distirbution
b	Brick distirbution
d_{next}	Next data distributino
$sext_x$	Stencil extents in x
$sext_y$	Stencil extents in y
s	Stencil consists $sext_x * sext_y$
S	Surface of a brick/block
V	Volume of a brick/block
H	Heigth in pixel
W	Witdh in pixel
PC	Pixel count
LPC	Local pixel count
RPC	Remote pixel count
SO	StencileCommunication overhead
C	Cost function
$R_{S/V}$	Surfacet to volume ratio
$R_{H/W}$	Higth to width ratio
o	Operation to generate workload
u	Number of used units
N	Number of used nodes

List of Figures

1.1	The dimensionality of the multi-temporal <i>RS</i> data.	2
2.1	Schema of von Neumann Computer Model	8
2.2	Growth in processor performance over 40 years [HP11]	8
2.3	Simple abstract schema of a task distribution on distributed system with four nodes.	10
3.1	A <i>SAR</i> image segmented in 20 data bricks with a size of $2048 \cdot 2048$ pixels each.	15
3.2	$3 \cdot 3$ stencil operation on a small array.	15
3.3	Different <i>DASH</i> patterns with 8 <i>units</i> on a segmented image in $8 \cdot 8$ bricks. Each color shade represents a block on an different <i>unit</i>	17
3.4	Multi-channel <i>RS</i> processing workflow	18
3.5	Different stencil operations: (ST1) with high extent in dimension 1, (ST2) stencil with high extent in dimension 0 and (ST3) stencil with small and similar extent in both dimensions. Color shade represent a <i>unit</i>	19
3.6	Visualization of an image segmented in $8 \cdot 8$ bricks loaded into a <i>BB</i> , <i>BN</i> and <i>NB</i> patterns in <i>DASH</i> using eight <i>units</i> . The different block colors shades correspond to the different <i>units</i> and the brick number corresponds to the respective node. Red highlighted brick numbers are locally available on the disk.	20
3.7	Visualization of an image segmented in $8 \cdot 8$ and already loaded into a <i>DASH</i> array with <i>BB</i> pattern. The different block colors shades and numbers correspondent to the different <i>units</i> /nodes. Red highlighted brick numbers are available in the local memory.	21
4.1	Schema of image segmentation into bricks on the left side and example for a single stencil operation on the right side.	24
4.2	Random data distribution on 8 nodes. Each shade of gray represents an assigned node.	25
4.3	$SO(d, s)$ for the different patterns with <i>ST1</i> , <i>ST2</i> and <i>ST3</i> . Assumed 100% brick locality while initialization. Right side the layout examples with the stencils to show alignment. Bottom plot: $BN = NB$	28
4.4	$SO(d, s)$ for the <i>BB</i> with different stencil extents.	28
4.5	Filtered $SO(d, s)$ for the different <i>units</i> with <i>ST3</i> . Assumed 100% brick locality while initialization. $BN = NB$	29
5.1	<i>HR-Cluster</i> Hardware overview	35
5.2	Comparison of read/write speed of local bricks with <i>NFS</i> on multi nodes. Green lines are summed average read/write performance of the single nodes. .	39

5.3	Average Initialization time on 3 nodes on the <i>HR-Cluster</i> in seconds by read image and copy to <i>DASH</i> with different brick localities, standard deviation displayed on top.	40
5.4	Example optimization result for 64 bricks distributed to 3 nodes and processed with 64 units. Each gray tone represents data located on the specific node.	43
5.5	Example optimization result for 64 bricks distributed to 3 nodes and processed with 64 <i>units</i> without considering brick locality. Each gray tone represents data located on the specific node.	44
5.6	Example optimization result for 64 bricks distributed to 3 nodes and processed with 64 <i>units</i> without considering brick locality using large stencils. Each gray tone represents data located on the specific node.	45
5.7	Average execution time in seconds of <i>ST1</i> , <i>ST2</i> and <i>ST3</i> with 2 nodes with implementation v1 on the <i>HR-Cluster</i>	46
5.8	Comparison of <i>BB</i> and <i>NB</i> pattern distribution on two nodes with eight <i>units</i> . Numbers display the node assignment of each brick. The block colors show the <i>unit</i> . The red lines mark borders where network communication is necessary.	47
5.9	$R_{S/V}$ ratio compare to the execution time in seconds of <i>ST2</i> with 3 nodes for the patterns <i>BB</i> and <i>NB</i>	48
5.10	Average execution time in seconds of <i>ST1</i> , <i>ST2</i> and <i>ST3</i> with 2 nodes with implementation v2 on the <i>HR-Cluster</i>	50
5.11	(1) Average time for copying a <i>DASH</i> array to the local buffer in seconds for the <i>ST1</i> , <i>ST2</i> and <i>ST3</i> stencil. The experiment facilitated implementation (v2) running on 2 nodes of the <i>HR-Cluster</i> . The confidence intervals show minimum and maximum copy times.	51
5.12	(2) Average time for calculating in seconds for <i>ST1</i> , <i>ST2</i> and <i>ST3</i> . The experiment facilitated implementation (v2) running on 2 nodes of the <i>HR-Cluster</i> . The confidence intervals show minimum and maximum calculation times.	51
5.13	(3) Average time for copying the local buffer to a <i>DASH</i> array in seconds for <i>ST1</i> , <i>ST2</i> and <i>ST3</i> . The experiment facilitated implementation (v2) running on 2 nodes of the <i>HR-Cluster</i> . The confidence intervals show minimum and maximum copy times.	52
5.14	Average execution time in seconds of <i>ST1</i> , <i>ST2</i> and <i>ST3</i> with 2 nodes with implementation v2 on the <i>SuperMUC-NG</i> . Confidence interval show min and max execution time.	53
5.15	Average execution time in seconds of <i>ST3</i> with 16 nodes with implementation v2 on the <i>SuperMUC-NG</i> . Confidence interval show min and max copy time.	54
5.16	(1) Average copy <i>DASH</i> to local buffer time in seconds of <i>ST</i> {500, 500}, 16 nodes and implementation v2 on the <i>SuperMUC-NG</i> . Confidence interval show min and max copy time.	55
5.17	<i>Weak/Strong scaling</i> shown on a single node and in multi node environments with implementation v1.	57
5.18	Weak scaling shown on four nodes divided for the different stencils with implementation v1.	58
5.19	Weak/Strong scaling shown on a single node and multi node environments with implementation v2 on the <i>HR-Cluster</i>	59

5.20	Weak/Strong scaling shown on the <i>SuperMUC-NG</i> with implementation v2 on 16 nodes divided for the different stencils.	59
5.21	Comparison of the amount of byte which are copied per <i>dash::copy</i> call between the different layouts [B = BLOCKED, N = NONE]	62
5.22	Calculated behavior of the <i>TCP</i> traffic on the network with a <i>MTU</i> of 1500. Top left: shows how many <i>TCP-Segments</i> are generated in total. Top right: show the utilization of the <i>MTU</i> size. Bottom left: the amount of data to communicate. Bottom right: shows the number of generated <i>TCP-Segments</i> per <i>dash::copy</i> call	63
5.23	Theoretically layout performance for ST2 on 2 nodes in the <i>HR-Cluster</i> with transposed data.	65

List of Tables

1.1	Satellite data center: the volume and velocity of RS data [MWW ⁺ 15], [Sen18], (*Projected based on [RMW ⁺ 18])	2
4.1	$R_{S/V}$ and $R_{H/W}$ for the given examples. [B = BLOCKED, N = NONE] . . .	25
5.1	Server in the DLR hardware configuration overview	36
5.2	<i>SuperMUC-NG</i> hardware configuration overview	36
5.3	Stencil Set 1: used in divers experiments as basis stencils. [N = None, B = Blocked]	43
5.4	Average execution times of 10 tests for the different stencils/layout combinations with implementation v1 on a 8000 * 8000 pixel image in the <i>HR-Cluster</i> . 43	
5.5	Stencil Set 2: larger stencils in comparison to set 1 to increase the stencils communication overhead <i>SO</i> . [N = None, B = Blocked]	45
5.6	Average cache-misses and standard deviation captured with <i>perf</i> with, 20.000 * 20.000 pixel image, with implementation v1. [N = None, B = Blocked]	49
5.7	Example execution times for both implementations on several different tasks with 2 nodes and a 20.000 * 20.000 pixel image. [N = None, B = Blocked] .	53

Bibliography

- [ABA⁺17] Emilio Arnieri, Luigi Boccia, Giandomenico Amendola, Chunxu Mao, Steven Gao, Tobias Rommel, Srdjan Glisic, Piotr Penkala, Milos Krstic, Anselm Ho, Uroschanit Yodprasit, Oliver Schrape, and Marwan Younis. A 60-channels adc board for space borne dbf-sar applications. In *International Symposium on Antennas and Propagation (ISAP)*. IEEE, Januar 2017.
- [Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring)*, pages 483–485, New York, NY, USA, 1967. ACM.
- [Apa19] Apache. Apache Hadoop. Website, 2019. [last visited 02. Dezember 2019].
- [Apa20] Apache. Apache Spark - Lightning-fast unified analytics engine. Website, July 2020. [last visited 5. January 2020].
- [BAZ12] Nicolas Baghdadi, Maelle Aubert, and Mehrez Zribi. Use of TerraSAR-X Data to Retrieve Soil Moisture Over Bare Soil Agricultural Fields. *IEEE Geoscience and Remote Sensing Letters*, 9:512–516, 05 2012.
- [BCPS13] Mamta Bhojne, Abhishek Chakravarti, A Pallav, and V Sivakumar. High performance computing for satellite image processing and analyzing—a review. *International Journal of Computer Applications Technology and Research*, 2:424 – 430, 07 2013.
- [BDK⁺17] Ujwala Bhangale, Surya S. Durbha, Roger L. King, Nicolas H. Younan, and Rangaraju Vatsavai. High performance gpu computing based approaches for oil spill detection from multi-temporal remote sensing data. *Remote Sensing of Environment*, 202:28 – 44, 2017. Big Remotely Sensed Data: tools, applications and experiences.
- [BFS89] William Bolosky, Robert Fitzgerald, and Michael Scott. Simple but effective techniques for NUMA memory management. *ACM SIGOPS Operating Systems Review*, 23(5):19–31, 1989.
- [BKBB18] P. Bastian, D. Kranzlmüller, H. Brühlchle, and M. Brehm. *High Performance Computing in Science and Engineering*. Bayerische Akademie der Wissenschaften, 2018.
- [Bok87] Shahid H. Bokhari. *Assignment Problems in Parallel and Distributed Computing*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

- [Bon00] André B Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203, 2000.
- [CCZ07] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.
- [CGH94] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The mpi message passing interface standard. In Karsten M. Decker and René M. Rehmman, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 213–218, Basel, 1994. Birkhäuser Basel.
- [ETS14] H. Edwards, Christian Trott, and Daniel Sunderland. Kokkos: Enabling many-core performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74, 07 2014.
- [FF16] Tobias Fuchs and Karl Fuerlinger. Expressing and exploiting multi-dimensional locality in dash. In *Software for Exascale Computing*, volume 113, pages 341–359, 01 2016.
- [FFK16] K. Fuerlinger, T. Fuchs, and R. Kowalewski. Dash: A c++ pgas library for distributed data structures and parallel algorithms. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 983–990, Dec 2016.
- [GHD⁺17] Noel Gorelick, Matt Hancher, Mike Dixon, Simon Ilyushchenko, David Thau, and Rebecca Moore. Google earth engine: Planetary-scale geospatial analysis for everyone. *Remote Sensing of Environment*, 202:18 – 27, 2017. Big Remotely Sensed Data: tools, applications and experiences.
- [GMÜ⁺17] N. Gorelick, M. Marconcini, S. Üreyen, J. Zeidler, V. Svaton, and T. Esch. Mapping the Urban Side of the Earth- the new GUF+ Layer. In *AGU Fall Meeting Abstracts*, volume 2017, pages IN51H–05, Dec 2017.
- [Gro19] HDF Group. The HDF5 Library & File Format. Website, November 2019. [last visited 20. November 2019].
- [HJ13] Pascal Hitzler and Krzysztof Janowicz. Linked data, big data, and the 4th paradigm. *Semantic Web*, 4(3):233–235, 2013.
- [HK14] Richard D Hornung and Jeffrey A Keasler. The raja portability layer: overview and status. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2014.
- [HP11] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2011.

- [HSHH15] Jasim Hiba, Ammar Hameed Shnainm, Sarah Hadishaheed, and Azizahbt Haji. Big Data and Five Vs Characteristics. *International Journal of Advances in Electronics and Computer Science*, 2(1):16–23, 01 2015.
- [IBM19] IBM. IBM - Spectrum Scale Overview. Website, July 2019. [last visited 28. November 2019].
- [Int19] Intel. Technical Details: INTEL XEON GOLD-PROZESSOR 6212U. Website, July 2019. [Online; last visited 15. July 2019].
- [ISO12] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, February 2012.
- [JDT⁺15] Jennifer Jay, Francisco Delgado, Jose Torres, Matthew Pritchard, Orlando Macedo, and Victor Aguilar. Deformation and seismicity near sabancaya volcano, southern peru, from 2002-2015. *Geophysical Research Letters*, 42, 03 2015.
- [LGP⁺11] C. A. Lee, S. D. Gasster, A. Plaza, C. Chang, and B. Huang. Recent developments in high performance computing for remote sensing: A review. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 4(3):508–527, Sep. 2011.
- [LM19] MNM-Team LMU Munich. DASH-Project. Website, November 2019. [last visited 20. November 2019].
- [LSBBH11] J. M. Lopez-Sanchez, J. D. Ballester-Berman, and Irena Hajnsek. First Results of Rice Monitoring Practices in Spain by Means of Time Series of TerraSAR-X Dual-Pol Images. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 4(2):412–422, Juni 2011.
- [MML17] Mabule Mabakane, Daniel Moeketsi, and Anton Lopis. Scalability of DL POLY on High Performance computing platform. *South African Computer Journal*, 29, 12 2017.
- [MWW⁺14] Yan Ma, Haiping Wu, Lizhe Wang, Bormin Huang, R. Ranjan, Albert Zomaya, and Wei Jie. Remote sensing big data computing: challenges and opportunities. *Future Generation Computer Systems*, 51, 11 2014.
- [MWW⁺15] Yan Ma, Haiping Wu, Lizhe Wang, Bormin Huang, Rajiv Ranjan, Albert Zomaya, and Wei Jie. Remote sensing big data computing. *Future Gener. Comput. Syst.*, 51(C):47–60, October 2015.
- [oBC19] University of British Columbia. MPICH — High-Performance Portable MPI. Website, November 2019. [last visited 20. November 2019].
- [PDCK11] A. Plaza, Q. Du, Y. Chang, and R. L. King. High performance computing for hyperspectral remote sensing. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 4(3):528–544, Sep. 2011.
- [Pro19] Prometheus. Top 500 - the list. Website, 2019. [last visited 20. November 2019].

- [Rec19] Leibniz Rechenzentrum. SuperMUC-NG - System Description. Website, 2019. [last visited 10. Dezember 2019].
- [RMW⁺18] Achim Roth, Ursula Marschalk, Karina Winkler, Birgit Schaettler, Martin Huber, Isabel Georg, Claudia Kuenzer, and Stefan Dech. Ten Years of Experience with Scientific TerraSAR-X Data Utilization. *Remote Sensing*, 10:1170, 07 2018.
- [SBL⁺17] Peter Strobl, Peter Baumann, Adam Lewis, Zoltan Szantoi, Brian Killough, Matthew Purss, Max Craglia, Stefano Nativi, Alex Held, and Trevor Dhu. The six faces of the data cube. In *Proc. Conf. on Big Data from Space (BiDS'17)*, pages 28–30, 2017.
- [Sen18] Sentinel-2 Team. Sentinel-2 Mission Status Report 153. Technical report, European Space Agency, 2018.
- [SKF18] Joseph Schuchart, Roger Kowalewski, and Karl Fuerlinger. Recent experiences in using MPI-3 RMA in the DASH PGAS runtime. In *Proceedings of Workshops of HPC Asia*, pages 21–30, 2018.
- [SSPP11] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and West Pomeranian. Impact of system and cache bandwidth on stencil computations across multiple processor generations. In *Proceedings of the Workshop on Applications for Multi-and Many-Core Processors (A4MMC) at ISCA*, volume 3, page 2, 2011.
- [UPC05] UPC Consortium. UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.
- [Vel16] Domenico Velotto. *Oil spill and ship detection using high resolution polarimetric X-band SAR data*. PhD thesis, Technische Universität München, 2016.
- [von93] J. von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.
- [WMZ⁺15] L. Wang, Y. Ma, A. Y. Zomaya, R. Ranjan, and D. Chen. A parallel file system with application-aware data layout policies for massive remote sensing image processing in digital earth. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1497–1508, June 2015.
- [YBC⁺07] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32, 2007.
- [YIL⁺13] Fumio Yamazaki, Yoji Iwasaki, Wen Liu, Takashi Nonaka, and Tadashi Sasagawa. Detection of damage to building side-walls in the 2011 Tohoku, Japan earthquake using high-resolution TerraSAR-X images. In Lorenzo Bruzzone, editor, *Image and Signal Processing for Remote Sensing XIX*, volume 8892, pages 299 – 307. International Society for Optics and Photonics, SPIE, 2013.

- [YSP⁺98] Katherine A. Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul N. Hilfinger, Susan L. Graham, David Gay, Phillip Colella, and Alexander Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.
- [ZKD⁺14] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. UPC++: A PGAS Extension for C++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1105–1114, May 2014.
- [ZMB⁺16] M. Zink, A. Moreira, M. Bachmann, B. Bräutigam, T. Fritz, I. Hajnsek, G. Krieger, and B. Wessel. TanDEM-X mission status: The complete new topography of the earth. In *2016 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, pages 317–320, July 2016.